UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Improving CPU performance modelling in QEMU

by

David Mahmoodi, Rafal Stepuch,
Ricardo da Silva, Aditya Tandon, Nayaab Gupta

A group design project report submitted for the award of
Master of Engineering

Supervisor: Denis A Nicole, Prof. Steve R Gunn
Examiner: Prof. Mark Zwolinski

December 12, 2013

ABSTRACT

QEMU is a machine emulator and virtualiser, and is able to run operating systems or programs written for one CPU, such as MIPS, on a completely different CPU. However, QEMU does not actually model any hardware, and thus cannot be used to perform CPU cache profiling or analysis. This report presents the implementation of a flexible cache model in the QEMU MIPS target, which is capable of emulating L1 and L2 instruction and data caches, and can be configured with various cache sizes and types. It is designed to run as fast as possible, since anything added to QEMU inevitably slows down execution. To further improve performance, a slimmed down Linux Kernel was built. This report also describes a set of analysis tools that were created in order to facilitate analysis of the data provided by the cache model. The results of analysing several GNU applications with these tools are presented in the final sections of the report.

Contents

List of Figures

List of Tables

Nomenclature and Abbreviations

2D	Two Dimensional
3D	Three Dimensional
ASCII	American Standard Code for Information Interchange
CPU	Central Processing Unit
CSV	Comma-Separated Values
D-Cache	Data Cache
DM	Direct Map
I-Cache	Instruction Cache
L1	Level One (refers to cache level)
L2	Level Two (refers to cache level)
LFU	List Frequently Used
LRU	Least Recently Used
MMU	Memory Management Unit
NOP	No Operation
OS	Operating System
TB	Translation Block
TCG	Tiny Code Generator
TLB	Translation Look-aside Buffer

Chapter 1

Introduction

1.1 Goal Specification

At the highest level, the goal of this project is to add MIPS CPU cache modelling support to the open source tool 'QEMU', and then to perform cache profiling and analysis using the system implemented. This and other important goals are outlined here.

1.1.1 Cache Support for MIPS Backend

The first goal of this project is to add cache modelling support to the MIPS back-end in QEMU. In particular, C code must be written that is integrated into QEMU and is designed to be as fast as possible. The code must be capable of modelling L1 and L2 instruction and data caches, and must model different cache architectures, specifically, direct mapped and two-way/four-way set associative caches, as well as different line lengths. In this context, 'cache modelling' means that the code should model the cache sufficiently such that cache profiling and analysis may be performed using it. Thus, in addition to modelling caches, the system also needs to be easily customisable, preferably with command-line arguments, and must provide data in a sensible, easily analysed format.

1.1.2 Cache Profiling and Analysis

Once cache modelling is added to QEMU, the goal is to use this system to perform a variety of cache analysis and profiling tasks designed to discover how cache performance could be improved. This should look at factors including the ratio of cache hits to misses, cache misses per line/instruction, and whether any areas of memory are particularly stressed. In addition, this should look at how varying cache design parameters, such as different combinations of L1 and L2 sizes and types, affects the performance. Conclusions

should be produced that look at how the cache design may be optimised, or how code in the kernel or applications could be improved.

1.1.3 Kernel Configurations and GNU Applications

In order to achieve the previous goal, some time needs to be spent researching into MIPS Linux kernel tools and GNU applications that are useful for performing cache profiling and stress testing. In addition, as QEMU is an emulator, it invariably runs relatively slowly, and performing the profiling on a standard kernel configuration would take far too long. Therefore, another goal is to use a build tool such as Buildroot[6] to configure and compile a slimmed down version of the Linux kernel, and the required applications, that will run at an acceptable speed.

1.2 Motivation

Most modern processors (including MIPS) have a cache available to them, which is essentially a piece of fast memory that is used by the CPU to store recently used data blocks from main memory. The main memory, which is larger than the cache, usually has much longer access times, and thus caches are designed to boost performance by reducing the number of slow memory accesses that the CPU must make. Cache performance is affected both by their design, and by the software that is running on the CPU and causing the cache to be accessed – these are both very variable parameters. Thus, it is desirable to maximise cache performance, both by improving software design by using cache profiling, and by improving the design of the cache itself. Having a tool to help perform both these tasks is very beneficial to MIPS engineers.

QEMU is a fast machine virtualiser/emulator which can be used to emulate an entire system, for example, GNU/Linux, without the need for actual hardware. This is very useful to MIPS engineers and developers as it allows for rapid system prototyping and debugging, without all the costs and headaches that come with developing on real hardware. Currently however, QEMU does not have any cache modelling support which means it cannot be used to do cache design analysis or profiling. The project's main goal is therefore to enhance this tool, and help drive development of MIPS hardware and software.

Chapter 2

Cache Design

2.1 An overview on caches

Caches are small, fast pieces of memory that are used to boost performance in a processor [11, 13]. Larger pieces of memory have high latency and access times (time used to retrieve data from memory) and caches are used to compensate for this. This creates a memory hierarchy (see Figure 2.1) as there is more than one piece of memory used to hold data [11, 13, 17]. Usually, memory close to the processor is small and fast as it should occupy minimum area and have quick access times whilst size of memory increases as distance from the processor increases [13]. Access times increase as well due to increasing size [13].

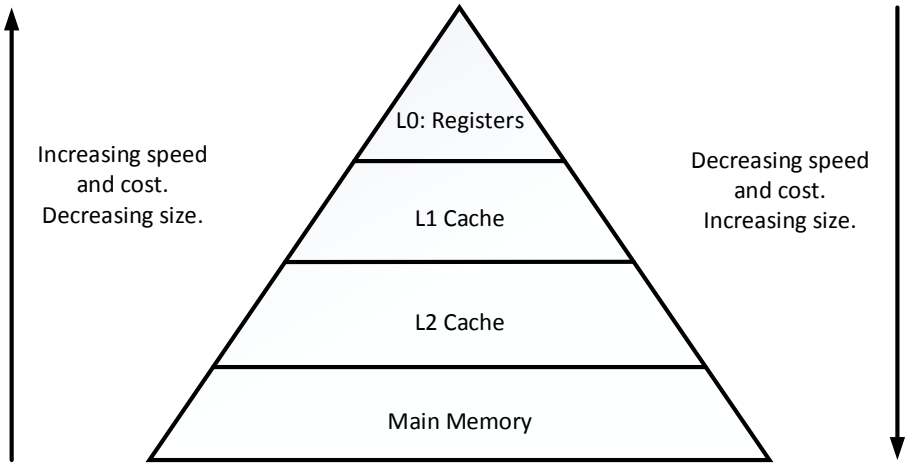


Figure 2.1: Memory Hierarchy (adapted from Memory Hierarchy [15]).

The principle of locality is the basis on which caches work [13, 17]. There are two types of locality which are discussed below [13, 17]:

- **Temporal Locality:** A piece of data which is referenced can be used again soon.
- **Spatial Locality:** Data blocks next to the referenced piece of data can be referenced soon.

Caches store data that is frequently used and also data that is close to the data being used. This way caches make use of both temporal and spatial locality [13, 17]. If the requested data is present in the cache, it is termed as a *hit* otherwise a *miss* occurs and the data needs to be recovered from the next piece of memory in the hierarchy (assuming it is present there) [13].

Caches can be configured as direct mapped, n-way set associative or fully associative as shown in Figure 2.2.

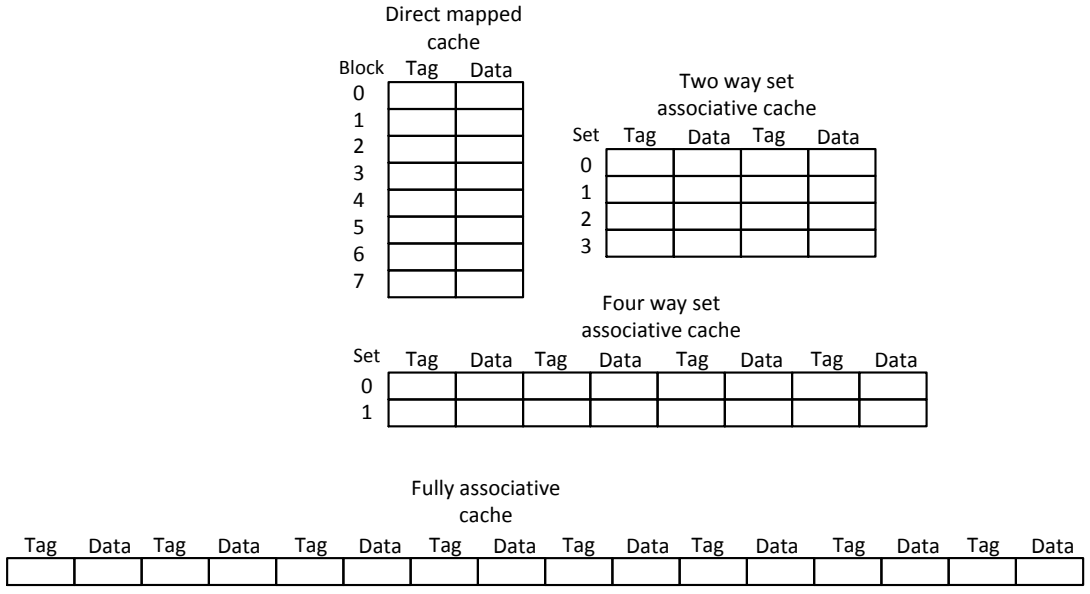


Figure 2.2: Cache types (reproduced from Patterson and Hennessey [13]).

It can be seen that in a direct mapped cache, data is stored in one location and in a fully associative cache data can be stored anywhere. A set associative cache is a compromise between the other two designs as data blocks are grouped into sets [13]. So in a two way set associative cache, data can be stored in two locations as one set has two cache lines available. These designs are later explained in detail in Section 2.5.

2.2 Cache read/write strategies

Caches have different strategies based on if data is present in the cache or not. The operation is different for reads and writes. The sections below discuss the different strategies that can be used.

2.2.1 Cache read strategy

On a hit, data is simply read from the relevant cache line. If a miss occurs then data needs to be retrieved from the other levels of the memory hierarchy depending on where it is present [13]. For example, if there is a miss in the L1 cache then the processor stalls and the address is sent to the L2 cache to retrieve the data. Data is then loaded into the L1 cache from the L2 cache and is read again and this time a hit will occur as the relevant cache line would have the requested address [13, 17].

2.2.2 Cache write strategy

A write hit occurs if the address that is being written to is present in the cache [17]. The write hit strategies are:

- **Write-through:** In this strategy data is written to both the cache and main memory. So all levels of the memory hierarchy get updated on a write. This keeps the cache coherent with other levels of memory and ensures that there is no stale data present anywhere in the memory hierarchy [13, 14, 17]. Figure 2.3 portrays this behaviour.

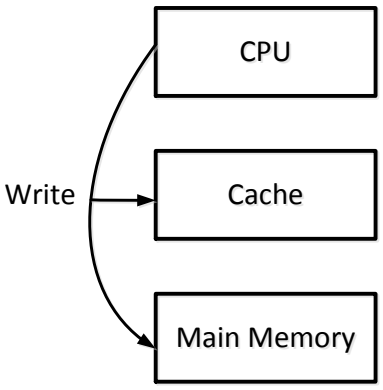


Figure 2.3: Write through cache.

The downside to this strategy is that writing to larger pieces of memory involves high latency and this strategy could take up a lot of time thereby making the cache in-effective in terms of speed [14].

- **Write-back:** Here, data is only written to the cache. The other levels in the memory hierarchy are not updated on a write. This saves time but now data is not consistent across the memory chain [13, 14, 17]. A cache line that is inconsistent with other levels of memory is deemed dirty and a dirty bit is added to a cache line to keep track of inconsistencies [13]. If a dirty cache line has to be replaced

then it first has to be written to main memory or the other levels of memory and then the line is replaced with the new value [13]. So this strategy saves time but is more complex to implement. Figure 2.4 shows how a write back cache operates.

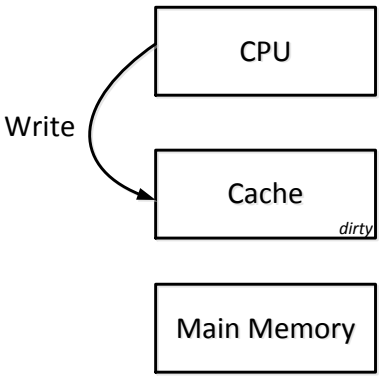


Figure 2.4: Write back cache.

If the block that is being written to does not exist in the cache then a write miss occurs [13, 14]. The following strategies are used to cope with a write miss:

- **Write Allocate:** On a write miss, the relevant cache block is first loaded from main memory and then the write is performed. The processor has to be stalled in this process [14]. Figure 2.5 demonstrates this strategy in operation.

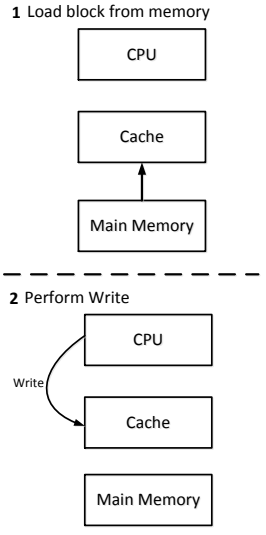


Figure 2.5: Write allocate strategy.

- **No Write Allocate/Write Around:** On a write miss, the data that needs to written to the cache is written to the main memory instead [11, 14] as shown in Figure 2.6 The correct value can be read when a read miss is encountered.

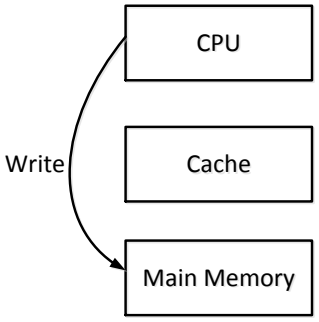


Figure 2.6: Write around strategy.

2.3 Cache Replacement Strategies

A cache replacement strategy is the algorithm used to determine which line in the cache needs to be discarded when the cache is full [13]. An efficient algorithm could increase the amount of hits in the cache as the data that is being frequently used is preserved in the cache. These strategies are used by set-associative or fully associative caches as there is more than one block of data in a set. Replacement strategies depend on the context of the application and many strategies exist for different applications. Only the strategies used in this project will be discussed and are listed below:

- **Least Recently Used (LRU):** In this strategy, the least recently used item is discarded first. To keep track of this, every cache line is assigned an extra bit which indicates if it was recently used [11, 13]. Usually, a 1 indicates the line was used recently and a 0 that it has not been used. The bits corresponding to each line in a set are updated whenever a particular set is accessed and are periodically cleared.

- **Random Replacement:** In this policy a cache line is randomly replaced to make room for the new data. No history bits are required for this approach and it is easier to implement but it does not always guarantee optimum performance [11, 13].

- **Least Frequently Used (LFU):** This strategy counts how many times a particular line in a cache is used [1]. So on replacement the value of the counter is compared and the line which has a counter with a lesser value is discarded. This strategy might not always correctly depict how often a cache line is being used as sometimes a particular line might be used repeatedly but after a certain period it might not be used at all. So at this point the counter associated with the line has a high value so it would seem that this particular line is still being used when it is not [1].

2.4 MIPS cache operations

The MIPS cache instruction pre-defines certain operations on the caches present in the system. Based on how the instruction is encoded, different operations can be performed on any of the caches present [12]. The cache instruction is shown in Figure 2.7.

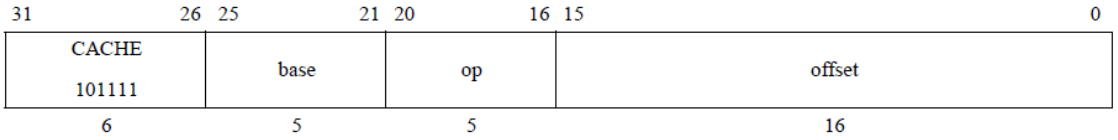


Figure 2.7: Cache instruction encoding (Cache Instruction [12]).

The offset is 16 bits long and is sign extended and added to the value in the base register to form an address [12]. The op field describes what kind of operation is to be performed and it also identifies which cache to operate on. The upper 3 bits specify the operation and the bottom two bits indicate which cache to perform the operation on [12]. The decoding of the op field is shown in Table 2.1. Bits[17:16] identify the type of cache.

Bits[17:16]	**Cache Type**
0	I-Cache
1	D-Cache
2	L3 Cache (not used)
3	L2 Cache

Table 2.1: Decoding op field to select relevant cache [12].

Once the cache is identified, bits[20:18] specify what kind of operation is to be performed on the cache. There are three different types of cache operations and they are listed below [12]:

- **Hit-type operation:** A cache operation is carried out if the address presented to the cache results in a hit. If a miss occurs then nothing happens.
- **Address-type operation:** This is processed as a normal cache operation. An address is checked for a hit or miss. If there is a hit then the operation is performed. On a miss, the relevant address is loaded from memory into the cache and then the operation is carried out.
- **Index-type operation:** Index operations are used to address the cache but there is no comparison of tags to check if a line is loaded. In such operations the way bit in the address field is used to identify a line within a set. The encoding is shown in Figure 2.8

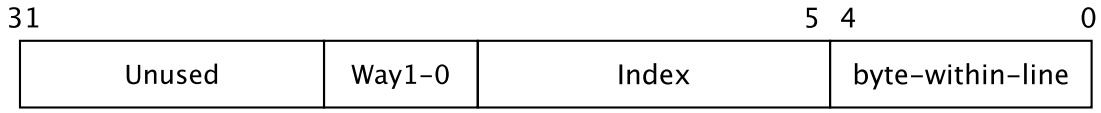


Figure 2.8: Index type operation [12].

Depending on the associativity the way bits can point to a line within a particular set to perform an index operation on [12]. The size of the way field is dependent on the associativity of the cache so, for example, a two set associative cache has one bit for the way field.

The list of cache operations are described in Table 2.2.

Bits[20:18]	**Operation**	**Description**
0	Index Invalidate	Sets the cache line at the specified index to invalid
1	Index Load Tag	Reads the tag from the specified cache index to the TagLo and TagHi registers
2	Index Store Tag	Stores tag from the TagLo and TagHi registers into the specified cache index
3	NOP	NOP
4	Hit Invalidate	If the cache line contains the address being matched then invalidate that particular cache line
5	Fill or Hit Invalidate	I-Cache: Fill operation is performed where the specified index is filled from address in main memory. Other Caches: Perform a hit invalidate.
6	Hit Writeback or NOP	For write-back caches, if there is a hit on the specified index, the cache line is written back to memory if it is dirty and the dirty bit is cleared. For a write-through cache this operation is a NOP.

7	Fetch and Lock	• If the cache does not contain the address then the particular address is fetched from memory and written to the cache. • The valid bit is set and another bit called the lock bit is set as well. The lock bit indicates that the particular cache line cannot be written to or replaced until the lock bit is cleared. • The lock state can be cleared by performing an index invalidate or a hit invalidate operation. • This operation is only for L1 caches (I-Cache and D-Cache).

Table 2.2: MIPS cache operations [12].

The operation defined as NOP for case 3 is an implementation dependent operation and in this project this can be ignored.

2.5 Cache Design

Caches included in this project are: direct mapped caches, two way set associative caches and four-way set associative caches. Cache parameters such as cache size, block size and associativity and the replacement algorithm could be varied to produce different designs as this would later help in performance analysis of different cache designs.

The sizes used for the L1 and L2 caches are shown in Table 2.3.

Size	**Block Size**	**Cache Lines**	**Size**	**Block Size**	**Cache Lines**
L1 Cache			L2 Cache		
2kB	1	512	256kB	4	16384
	2	256		8	8192
	4	128		16	4096
	8	64		32	2048
4kB	1	1024	512kB	4	32768
	2	512		8	16384
	4	256		16	8192
	8	128		32	4096
8kB	1	2048	1MB	4	65536
	2	1024		8	32768
	4	512		16	16384
	8	256		32	8192
16kB	1	4096	2MB	4	131072
	2	2048		8	65536
	4	1024		16	32768
	8	512		32	16384
32kB	1	8192	4MB	4	262144
	2	4096		8	131072
	4	2048		16	65536
	8	1024		32	32768

Table 2.3: Cache size configurations.

The other design choices are listed below:

- **Virtually Indexed and physically tagged cache:** L1 caches in MIPS are indexed using the virtual index from the virtual address but are tagged using the physical address [16]. This helps to run the cache faster as address translation in the TLB can be done simultaneously with cache indexing. When the index is

found, the tag in the cache is compared against the tag produced on translation from the TLB and a hit/miss is produced [13, 16]. This operation is depicted in Figure 2.9.

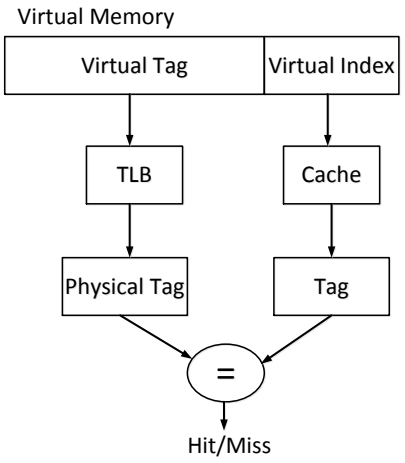


Figure 2.9: Virtually indexed physically addressed cache.

This approach could lead to cache aliasing where the same memory location is cached in different cache locations. This occurs because different virtual addresses describe the same location in different cache indexes [13, 16].

Caches with sizes up to 4kB with a direct mapped implementation undergo no aliasing as the virtual index is same as the physical index. This is because addresses are translated in 4kB pages [16]. Similarly, a 16-kB four-way set associative cache would not undergo aliasing as each index points to more than one cache line and so the virtual index would be the same as the physical index [16]. Bigger cache sizes are susceptible to aliasing and the operating system (OS) needs to be able to work around cache aliases to avoid potential bugs [16].

L2 caches are physically indexed and physically tagged as they require the complete physical address to handle a miss and also are too big to index using the virtual address [16]. They are not susceptible to cache aliasing as they use a physical address.

- **Split or Unified:** L1 caches in MIPS have always been split into an instruction cache (I-Cache) and a data cache (D-Cache) and this separates instructions from data and an instruction fetch will not conflict with a data store/load [13, 16].
  L2 cache is unified as it is a more complex, expensive operation otherwise [16]. The L2 cache is also bigger as compared to the L1 caches to keep miss rates low, but this also means it is slower than the L1 cache [13].

2.5.1 Direct Mapped Cache Design

The first cache design for the instruction and data cache is a direct mapped design. Here, every location from memory is mapped onto one cache location. A direct mapped structure is shown in Figure 2.10.

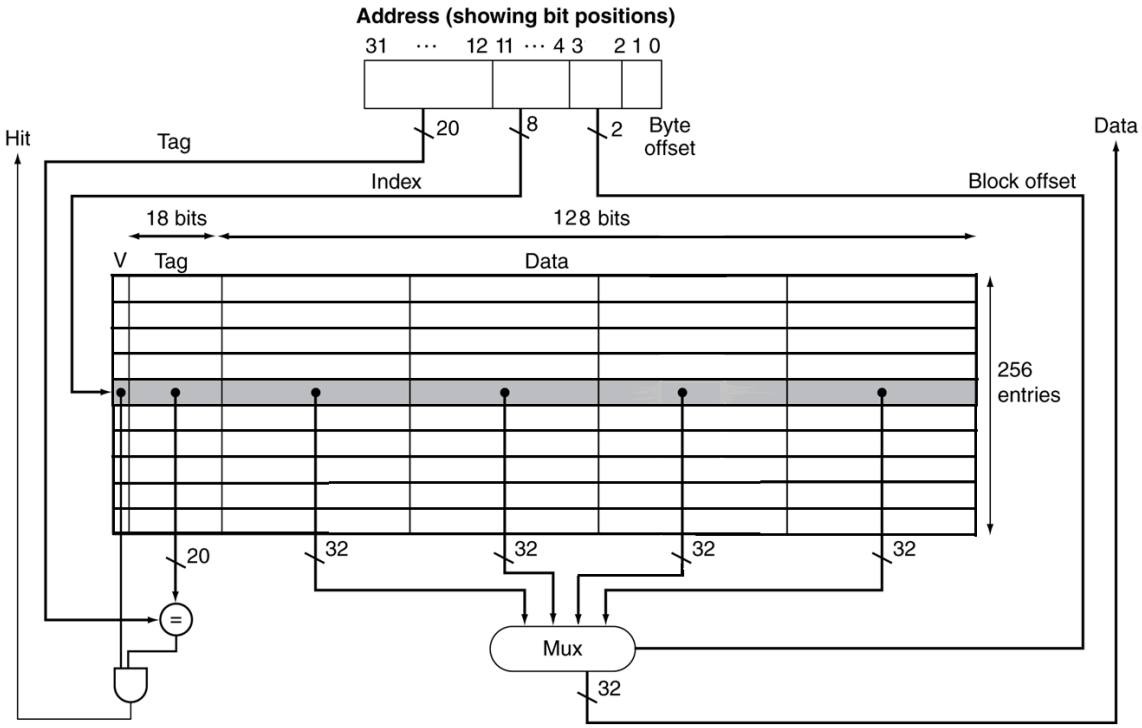


Figure 2.10: Direct mapped cache (adapted from Patterson and Hennessey [13]).

From Figure 2.10 it can be seen that only one line is compared for a hit or miss at any time. A cache line typically consists of a valid bit, the tag field and the data block [13]. The tag field is used to identify if the associated cache line contains the data requested by the processor. The valid bit indicates if the information present in the cache line is correct. To make use of spatial locality, data in a cache is grouped into blocks and therefore a cache line contains more than one word of data. The word offset is used to identify which block of data is needed [13]. In addition to this, the line also has a lock bit which indicates if a particular line is locked (see 2.2) [16, 12]. Since the data is word aligned we ignore the first two bits of the address as that would indicate a byte offset. So, when in operation, the index field of the address identifies the cache line and then the tag from the address is compared against the tag in the cache line. If they match, there is a cache hit and the word offset is used to retrieve the data from the cache. Otherwise, there is a cache miss and data needs to be retrieved from the L2 cache or main memory [13].

A direct mapped design is a suitable for the L1 caches (I-Cache and D-Cache) as hit times are low and also the implementation is simple as compared to the other designs. The drawback is that for big programs, this design produces many misses as different

memory locations map onto the same cache location and therefore it can have a high miss rate [13].

2.5.2 Set Associative Cache Design

To improve performance a set associative cache can be implemented which makes more efficient use of the space available in the cache. A cache can be N-way set associative, that is, it can have n sets and data can be placed in N locations [13]. In this project, a two-way set associative cache and a four-way set associative cache have been designed and implemented.

Figure 2.11 shows a two-way set associative cache. It can be seen that a block of data can be placed in two possible locations. The cache is indexed by its set number and every line in the set has to be compared for a hit. Once the line that produces a hit is identified, the relevant data is outputted.

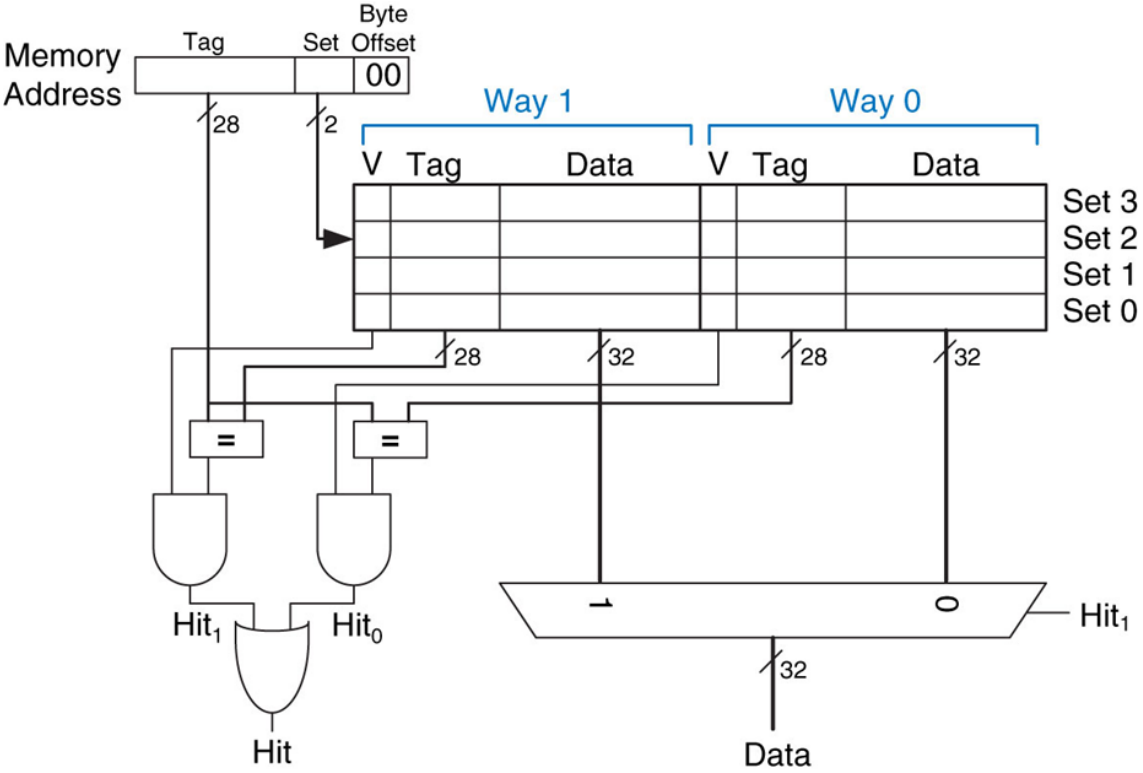


Figure 2.11: Two-way set associative cache (reproduced from Harris and Harris [10]).

The caches have a configurable replacement strategy and can use Least Recently Used (LRU), Least Frequently Used (LFU) or Random Replacement.

A set associative design is suitable for the L2 cache as miss rates need to be kept

to a minimum. Set associative caches can also be used for the design of L1 caches to compare performance with a direct mapped design. Although, finding a data block is not as straightforward as in a direct mapped cache as two or four lines have to be compared for a hit simultaneously depending if a two-way or four-way set associative cache design is used [11, 13, 16].

2.6 Algorithmic Design of Cache Operations

This section discusses the algorithmic approach taken in implementing cache operations in MIPS. As described previously, these cache operations differ with the type of cache involved (i.e. Direct Mapped or Set Associative) and the replacement strategy used (LRU, LFU or Random). It is important to note that the cache implemented in this project does not actually store the data that a real cache would hold. It only stores the information necessary to assess the cache performance – it needs to keep track of which tags are stored in the cache, their associated 'lock' and 'valid' bits, and it needs to keep track of the number of cache hits and misses per line.

2.6.1 Direct Mapped Caches

A simplified version of the cache structure used in the project is shown in Listing 2.1 (the full version is described in Section 7.4.2). Each line of the cache is allocated one of these structures, and thus the entire cache is essentially an array of them. The number of cache lines (and hence the size of the array) depends on the value of the index width of the cache. For example, an index width of 8 bits would correspond to $2^8 = 256$ cache lines.

```
struct cache_line {
  unsigned int tag   : TAG_WIDTH
  unsigned int valid : 1;
  unsigned int lock  : 1;
}

// Define the cache:
cache_line cache[1<<INDEX_WIDTH]
```

Listing 2.1: Simplified cache line data structure

2.6.1.1 Extracting Cache Fields

Figure 2.12 illustrates how a referenced 32-bit MIPS memory address is divided into a tag field of size 'tag width', an index field of size 'index width' and a two bit offset field.

The tag and index fields need to be extracted from a given referenced memory address in order to perform a cache lookup. To decode the tag, the memory address is simply shifted to the right by [index width + 2] bits. This shift amount is referred to as the 'tag shift'. The index is slightly more involved, and is extracted by first bit masking the top portion of the address, and then right shifting (by two bits) the remainder. The number of bits that must be masked depends on the index width – a wider index will require few bits to be masked. Pseudo code for extracting the two cache fields is shown in Listing 2.2.

```
index = (addr & INDEX_MASK) >> INDEX_SHIFT
tag   = addr >> TAG_SHIFT
```

Listing 2.2: Extraction of index and tag fields

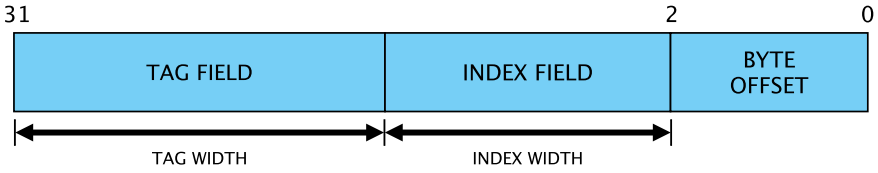


Figure 2.12: A referenced 32-bit MIPS address

2.6.1.2 Data Lookup

In this report, 'data lookup' refers to the action of attempting to retrieve data from a cache, which will result in a hit or a miss. In the direct mapped cache model, if the tag contained in the referenced address is present in the cache at the specified index then it is a hit, and the hit counter is incremented. If the tag is not present, or the valid bit is not set, or the line is locked, then it is a miss, and the miss counter is incremented. If the line is unlocked, the tag is then stored at the specified index location. This algorithm is presented in textual and flow chart form (see Figure 2.13) below.

```
1.      Extract the index value from the referenced memory address.
2.      Extract the tag value from the referenced memory address.
3.      Check if the extracted tag value is present at the extracted
        index location in the cache array.
4.      Check if the cache line referenced by the extracted index value
        is valid (valid bit is one).
5.      If conditions 3 and 4 are true then perform the following steps:
        a.      Display:    H i t !
        b.      Increment the hit counter.
        c.      Perform step 7.
6.      Otherwise perform the following steps:
        a.      Display:   M i s s !
        b.      Increment the miss counter.
        c.      Check if the cache line referenced by the extracted
                index value is locked.
        d.      If yes then perform the following steps:
                i.      Display   L i n e  Locked!
                ii.     Perform step 7.
```

```
        e.      Otherwise perform the following steps:
                i.      Store the extracted tag value at the index
                        location in the cache array.
                ii.     Set the valid bit of the cache line as one.
                iii.    Perform step 7.
7.      End.
```

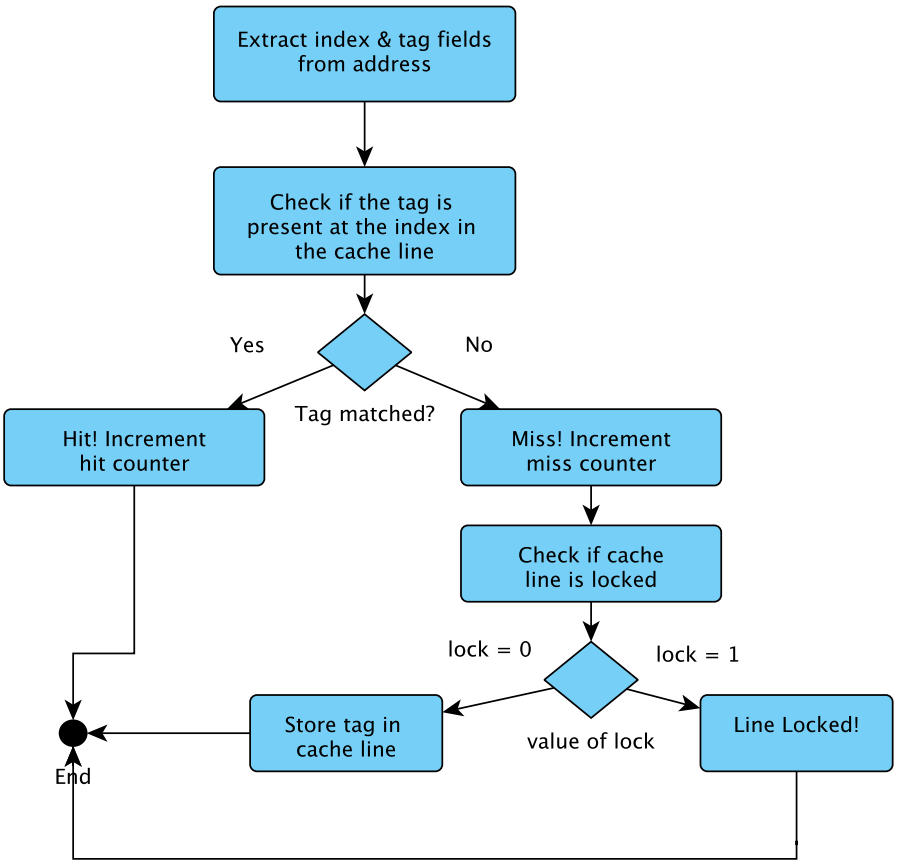


Figure 2.13: Flow chart illustrating the "data lookup" operation of a direct mapped cache

2.6.1.3 Cache Instructions

Most of the cache instructions are relatively easy to implement, and are outlined here.

Cache Invalidate

The valid and the lock bits of the cache line referenced by the extracted index value are set to zero as shown in the pseudo code below.

```
cache[index].valid = 0
cache[index].lock  = 0
```

Load Tag

The tag value present in the cache line referenced by the extracted index value is stored in a variable called *taglo*. This is shown in the pseudo code below.

```
taglo = cache[index].tag
```

Store Tag

The tag field extracted from the referenced memory address is stored in the cache line referenced by the extracted index value as shown in the pseudo code below:

```
cache[index].tag = taglo
```

Hit Invalidate

The algorithm for the hit invalidate function of a direct mapped cache is shown in Figure 2.14.

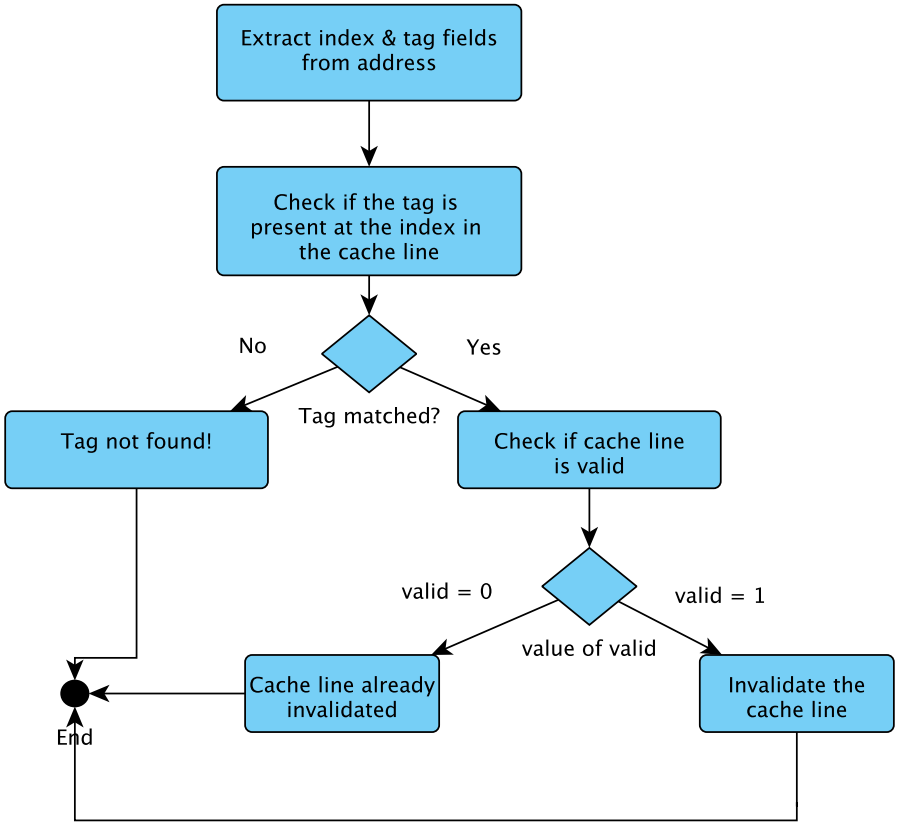


Figure 2.14: Flow chart illustrating the "hit invalidate" operation of a direct mapped cache in MIPS

Fill

The algorithm for the fill function of a direct mapped cache is shown in Figure 2.15.

Fetch and Lock

The algorithm for the fetch and lock function of a direct mapped cache is shown in Figure 2.16.

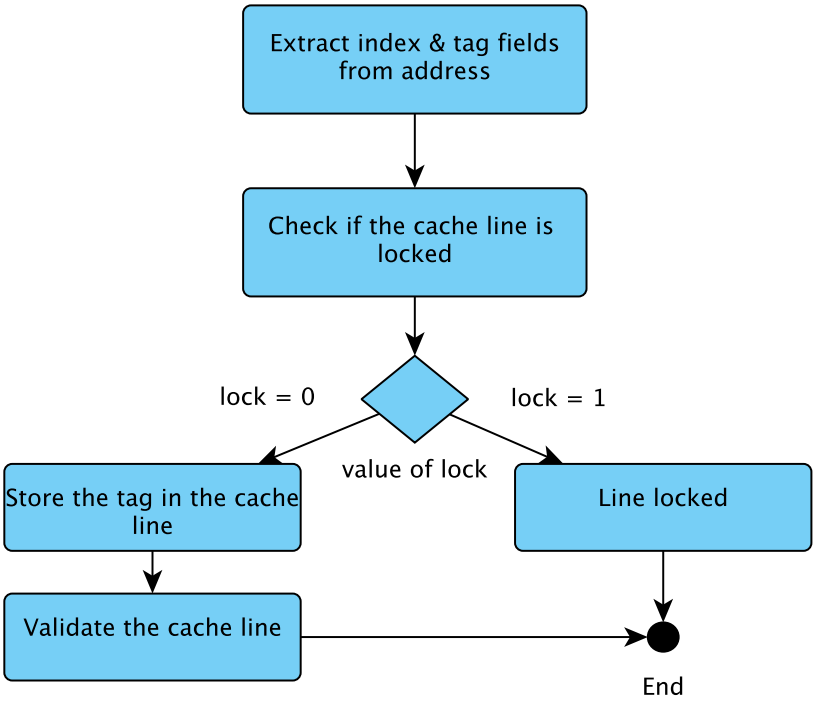


Figure 2.15: Flow chart illustrating the "fill" operation of a direct mapped cache in MIPS

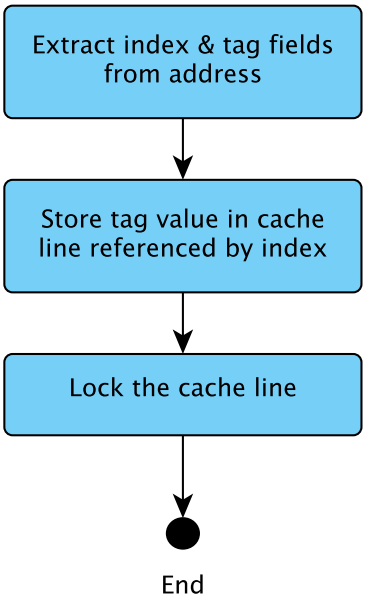


Figure 2.16: Flow chart illustrating the "fetch and lock" operation of a direct mapped cache in MIPS

2.6.2 Set Associative Caches

The (simplified, as before) structure of an N-way set associative cache is shown is Listing 2.3. One way of creating a set associative cache is to create N arrays of these

structures – one array for each 'way'. For example, a 2-way set associative cache can be implemented by declaring two arrays as shown. In the system actually implemented, each cache is stored as a single array, where the top half is one set, and the bottom half is the other. However, description of the algorithm is simplified if they are considered to be two separately named arrays.

```
struct cache_line {
  unsigned int tag           : TAG_WIDTH;
  unsigned int valid         : 1;
  unsigned int lock          : 1;
  unsigned int replacement_field : 1;
}

// Define the cache:
cache_item wau1[N_LINES];
cache_item way2[N_LINES];
```

Listing 2.3: Simplified cache line data structure (2-way set associative cache)

The *replacement_field* bit in the structure is used by the LRU replacement strategy – note that for a 4-way set associative cache, this would be 2 bits wide. Furthermore, if the cache was using the LFU replacement strategy, this field would be a counter, and if it was using Random replacement, this field would not be required at all.

N_LINES is equal to $2^{index\_width}/associativity$, so that the total number of lines is still 2^{index_width}, as in the direct mapped case. In addition, the method of extracting the tag and index fields is the same as for a direct mapped cache.

2.6.2.1 Data Lookup

In order to look up data in a set associative cache, more than one location must be checked (e.g., two locations, in a 2-way set associative cache). When a miss occurs, the cache controller must select which of the N *ways* should be replaced with the new data tag, according to the replacement strategy that is employed.

For a cache using the Least Recently Used replacement strategy, the *replacement_field* field is used to determine which of the possible locations was used least recently. Every time a line is used, its *replacement_field* field is set to the maximum possible value, and all the other possible lines have their *replacement_field* fields decremented (until they reach zero). Therefore, the least recently used line is the one with the lowest *replacement_field* field. If more than one line have the same value, then the first one is chosen for simplicity. For example, in a 2-way cache, the *replacement_field* can either be 1 or 0, where a line with 0 was used less recently.

Least Frequently Used is implemented in a very similar way, except that the *replacement_field* is a counter that is incremented every time a line is hit. Thus, the line with the lowest count is the one being used less frequently, and is chosen to be replaced with new data (and the counter set to 0 again). However, in order to prevent one line being rendered irreplaceable due to a high number of sequential hits, the counter may be reset to 0 periodically.

When the Random replacement strategy is used, the candidate *way* is selected by generating a random number. For example a 2-way set associative cache using random replacement will randomly generate 0 or 1 corresponding to the 'way 1' cache and 'way 2' cache respectively.

2.6.2.2 Cache Instructions

Cache Invalidate

The cache invalidate function is exactly the same as for the direct mapped cache, as the operation is given an 'exact' index – the index includes the cache set to use. There is no need for the algorithm to decide which cache line to use.

Load/Store Tag

Again, these functions are implemented in the same way the direct mapped versions, as they are also given an exact index.

Hit Invalidate

This function is different however, as the algorithm must look for the given tag in all possible locations. If the tag value is found in a valid cache line of one of the *ways*, then the cache line is invalidated. The algorithm for the hit invalidate function of a 2-way set associative cache is given below:

Fetch and Lock

For an N-way set associative cache, this function checks which *way* contains the extracted tag value at the referenced set index. If the tag value is found in one of the *ways*, then the cache line is locked. If it is not found, then one of the *ways* is replaced with the new data, according to the replacement algorithm used, and the line locked. Therefore, algorithmically this operation is identical to a data lookup, except that once the line has been chosen it is locked.

Chapter 3

QEMU functional description

This chapter explores QEMU in detail, and discusses the ways in which the customer requirements may be satisfied best. A high level description of QEMU will be given, and then the parts relevant to this project are looked at in more detail. In particular QEMU helpers, and the Tiny Code Generator will be explored. These sections are heavily based on information in the QEMU documentation[8] and in a paper the author of QEMU released about its design, [7].

3.1 Overview

QEMU is a fast machine emulator and virtualiser [7]. It is designed to run a guest system on a host system as efficiently and quickly and possible. This project uses QEMU solely as an emulator, specifically to emulate a MIPS system. In this context, "system" effectively refers to executable code, which can either be a single process, or an entire kernel with associated peripherals (network devices, storage drives, etc.). The first mode is known as "user mode", where a binary compiled for one CPU/architecture, such as MIPS, may be run on a different CPU, such as Intel x86. The second mode is known as "system mode", where an entire machine may be emulated, such as the MIPS Malta development board. However, it is important to note that QEMU does not actually emulate the guest CPU hardware. It executes guest instructions by dynamically translating them into host instructions, using a module called the Tiny Code Generator (TCG) which will be explained in more detail in Section 3.3 along with the way it performs dynamic translation. This is important because instead of implementing a "real" hardware cache, the project focusses on implementing a cache in software that is integrated into the QEMU system using TCG.

3.2 Targets

In QEMU, each CPU architecture that is emulated is known as a target, and has a top level folder dedicated to it in the QEMU source tree (such as 'target-mips' for MIPS CPUs). In this folder, the CPU is defined (cpu.h), a set of translation routines is defined (translate.c), and other target specific code is stored. Each target requires its own set of translation routines to allow TCG to process opcodes for that architecture, along with 'helper' code to help this process.

3.3 TCG

The QEMU Tiny Code Generator (TCG) (formerly known as 'dyngen' [7]) is the bridge between guest instructions and host instructions – it is essentially a compiler backend that performs the code generation and dynamic translation for QEMU. This section presents an overview of the parts of TCG relevant to the project; a more detailed description, along with information of all the TCG functions, is available in the QEMU documentation.

As mentioned previously, TCG is used to translate guest instructions to host instructions at run-time, and uses a form of 'dynamic translation' to do this (see Figure 3.1). Essentially, each guest instruction results in concatenated blocks of static host code that have been generated at compile time, and caches the resulting binary code. Thus it is faster than an interpreter, as the guest instructions only need to be decoded once, and it is simpler than a full code translator [7].

TCG defines a number of 'micro operations' (instructions) which are implemented in C, and compiled into object files. Translation is then a process of concatenating these functions into a single block which performs a required instruction (resulting in a 'Translated Block' – TB). A QEMU basic block is essentially a list of instructions that ends in a branch instruction. These blocks are placed in the TCG internal cache, which, along with a number of optimisations at compile and run time, also improves execution performance. This translation process is clearly target dependent – different CPU architectures with different instruction sets will require different translation routines. In QEMU these translation routines are all hand-coded and usually stored in a translate.c (or similarly named file) in a target directory.

In many cases TCG requires temporary variables to hold the state of something while translating an instruction. For example, in a load or store instruction, the required memory address is calculated from the base and offset specified in the opcode, and stored in a temporary variable. These variables are temporary in the sense that they are only live in the basic block in which they were declared. 'Local temporary' variables are also used, which are only live in the function (TB) in which they were declared.

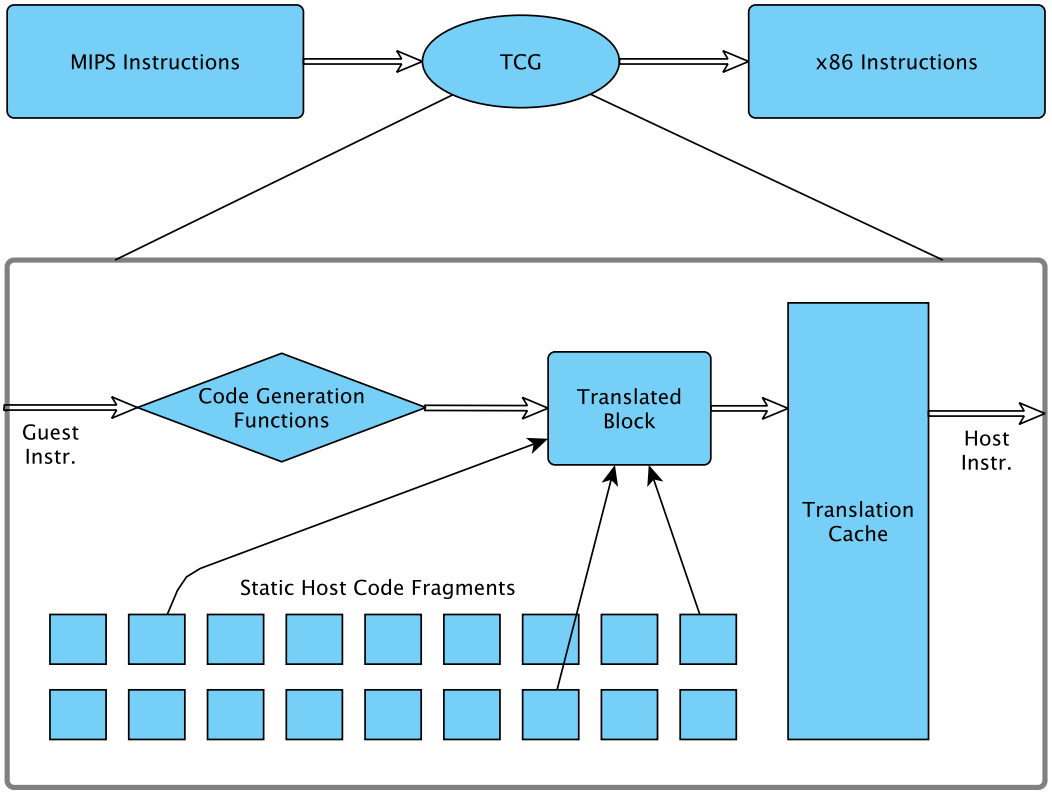


Figure 3.1: A simplified illustration of QEMU dynamic translation

3.4 TCG Helpers

One concept that needs to be made clear is that of helpers, in the context of QEMU target code generation. As explained in Section 3.3, TCG is used to translate guest instructions into executable host instructions. Essentially it does this by examining each opcode that is executed and calling the relevant code generation routine in translate.c. In many cases this is a simple matter, since there may be a one-to-one mapping between guest instructions and host instructions (e.g., a MIPS ADD instruction is very similar to the x86 ADD instruction). However, in other cases, TCG may not have primitives built in to handle the required operation, or the target has specific functionality that doesn't involve generating code. For example, the MIPS target has a special memory management unit (MMU) which essentially emulates a TLB required by MIPS processors. In this case, helpers provide a solution.

Helpers provide flexibility in the code generation, as they allow arbitrary function calls to be added to translation blocks; see Figure 3.2. TCG recommends using these to perform any complicated or infrequently used guest instruction, especially when the guest instruction would require more than twenty or so TCG instructions. They provide a surface on which to base the system implemented in this project, as they can be inserted into the flow of program execution, and hence used to emulate cache activity.

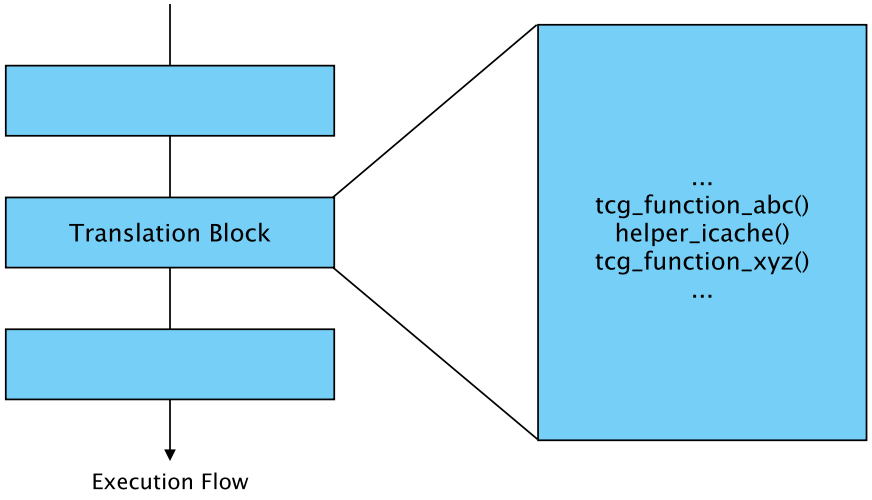


Figure 3.2: Helper code is inserted into TBs during code generation

The existence of the MIPS specific TLB is an added help, as it can be used as an example of how best to use helpers.

Chapter 4

Linux Kernel and Buildroot

As QEMU system mode requires an operating system to start, this chapter describes an easy way of compiling *Linux*. The tool used for that is called *Buildroot* and consists of a number of Makefiles configured via *menuconfig*. Being a robust and flexible tool, it allows for kernel and system configuration, toolchain compilation, root file system generation, as well as pre-installation of many software packages. The main reason for building the kernel from source rather than using an existing distribution (e.g. Debian) is its small size and simplicity. An operating system compiled by Buildroot is targeted mainly at embedded systems and provides only the basic system which runs much faster in QEMU than any pre-made solution. [6]

4.1 Downloading and Configuring Buildroot

This project uses Buildroot 2013.11 release. The files can be downloaded from `http://buildroot.uclibc.org/download.html`. There are three steps to follow before compilation (executed in Buildroot folder). The first one:

```
$ make qemu_mips_malta_defconfig
```

copies the default configuration for MIPS Malta board, which is already set up to work with QEMU. At this stage a working system can be compiled if no customisation is needed. The Linux kernel can be configured using:

```
$ make linux-menuconfig
```

This command opens a menu (Figure 4.1), which allows the user to choose many low-level system options as required for a specific task. In this project all default values are kept.

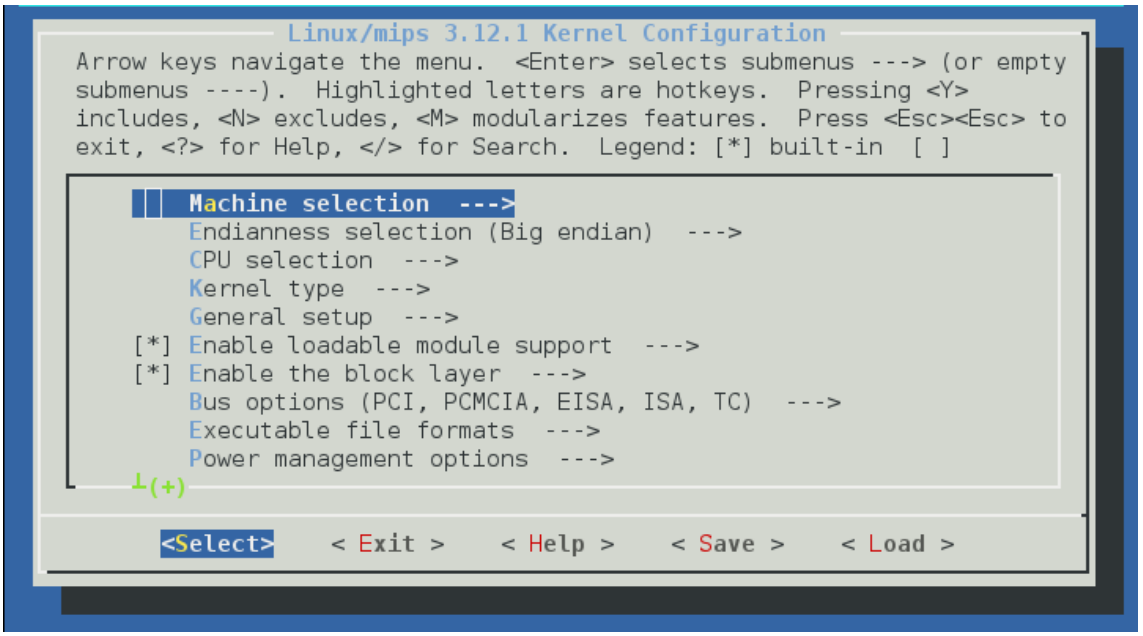


Figure 4.1: Linux kernel configuration via menuconfig

Other menu configuration (Figure 4.2), which is used more extensively in the project sets up Buildroot itself.

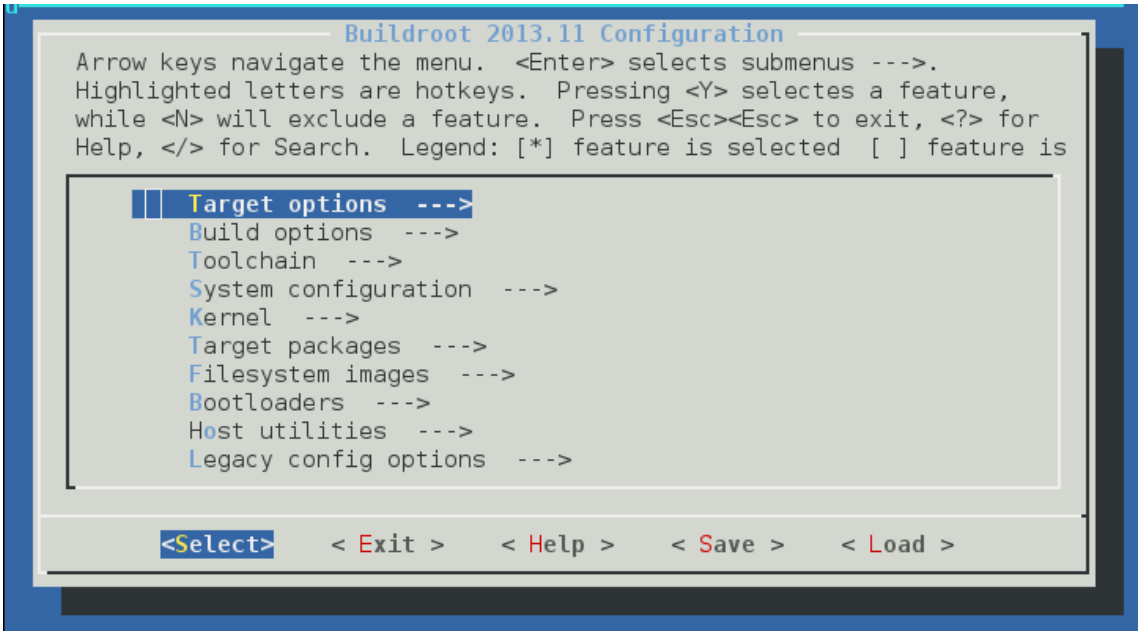


Figure 4.2: Buildroot configuration via menuconfig

It gives an opportunity to choose a compilation toolchain, a version of Linux kernel, a CPU architecture, filesystem images, target software packages etc. The changes made to the default configuration in this project include: different hostname, root password and extra packages selected. These packages are pre-installed and used to test cache performance:

- bzip2 — data compressing tool
- gnuchess — chess game
- imagemagick — image files converter and editor
- stress — tool imposing high load on the computer

Once Buildroot is configured, the compilation can be started by typing:

```
$ make
```

After successful compilation, the kernel image and the root file system can be found in *./output/images*. They are used for QEMU system mode.

4.2 MIPS Cross-Compiler

The MIPS toolchain, located in *./output/host/usr/bin*, can be used to compile programs for MIPS architecture. For instance, *./output/host/usr/bin/mips-buildroot-linux-uclibc-gcc* may be used to create a binary from a C file, which can then be run in QEMU user mode.

4.3 Configuring Linux for Automatic Cache Simulation

By default, Linux boots to *login* and a user has to log into the system. Then a test program can be executed and after that the system needs to be powered off. The problem is that this sequence will have to be repeated for each cache configuration that the user wants to test. Instead of that, Linux can be configured to automatically execute a program and exit without the need of logging in or explicitly calling the *poweroff* command.

4.3.1 Running Programs Automatically After Boot Sequence

Firstly, the root file system located in (*rootfs.ext2*) file needs to be mounted. Assuming there is a mount folder */mnt/guest*, this command should be used:

```
$ sudo mount path_to_rootfs/rootfs.ext2 /mnt/guest/
```

After that a shell script can be created on guest file system, which executes test programs. For instance:

```
$ nano /mnt/guest/home/default/run_test.sh
```

Any text editor can be used for this operation. It is imperative to add the *poweroff* command at the end of the file. The file needs to be made executable using:

```
$ chmod 755 /mnt/guest/home/default/run_test.sh
```

Listing 4.1 shows an example of *run_test.sh* script.

```
#!/bin/sh

## Login as default user -> disable when running a test
#echo "/bin/login -f default"
#exec /bin/login -f default

### ### Short tests ### ###

## Stress memory for 10 seconds
echo "stress --cpu 1 --io 1 --vm 10 --vm-bytes 8M --timeout 10s"
su default -c "stress --cpu 1 --io 1 --vm 10 --vm-bytes 8M --timeout 10s"

## Create 512kB of random data
#echo "dd if=/dev/urandom of=/dev/zero bs=524288 count=1"
#su default -c "dd if=/dev/urandom of=/dev/zero bs=524288 count=1"

## Exit the system -> enable when running a test
echo "poweroff"
poweroff
```

Listing 4.1: Sample test script

When the script is ready, it should run after the boot sequence. To achieve that, open */mnt/guest/etc/inittab* and modify the line:

```
ttyS0::respawn:/sbin/getty -L ttyS0 115200 vt100 -n
```

to

```
ttyS0::respawn:/sbin/getty -l /home/default/run_test.sh -L ttyS0 115200
    vt100 -n
```

By adding *-l* option to */sbin/getty*, the */home/default/run_test.sh* is executed instead of the login shell.

Save all the changes and unmount the drive:

```
$ sudo umount /mnt/guest/
```

4.3.2 Adding Extra Hard Drive

If programs, which are used in testing, operate on files, then the default free disk space on (*rootfs.ext2*) may not be sufficient. Creating an additional drive is easy but it requires

the *qemu-img* binary. Therefore, to perform this step QEMU needs to be compiled first (see Section 5.1).

To create new 50MB drive image use:

```
$ path_to_qemu_buid_directory/qemu-img create datafs.ext2 50M
```

The drive image needs to be formatted with:

```
$ mkfs.ext2 datafs.ext2
```

Once the drive is ready, it needs to be added to Linux *fstab* so that it is mounted automatically at boot. Firstly, mount *rootfs.ext2* as described in Section 4.3.1. Create a mount point on the root file system, e.g.:

```
$ sudo mkdir /mnt/guest/testdir
```

Now the */mnt/guest/etc/fstab* needs to be modified:

```
$ sudo nano /mnt/guest/etc/fstab
```

Add the following below the line starting with */dev/root*:

```
/dev/hdb        /testdir        ext2     defaults          0      2
```

This line automatically mounts */dev/hdb* drive to */testdir* folder when Linux boots. Save and close the file and unmount root file system file. When using the drive in QEMU, *datafs.ext2* should be used as *hdb* drive. To add data to this drive, mount it in the same way as *rootfs.ext2* to */mnt/guest* folder.

Chapter 5

User Guide

This chapter describes how to compile QEMU and how to use options implemented in this project. It also gives information about post-processing generated log files and problems that a user can encounter while running Linux on MIPS in QEMU. For more information about QEMU, please see the official documentation [3].

5.1 Configuring and Building QEMU for MIPS Guest Architecture

The latest code of QEMU, with changes introduced in this project, can be obtained from `https://github.com/rmhsilva/qemu`. Use either git:

```
$ git clone https://github.com/rmhsilva/qemu.git
```

or a direct link to a zip file of the repository:

```
$ wget https://github.com/rmhsilva/qemu/archive/master.zip
```

The code should not be compiled in QEMU top folder, therefore a build directory needs to be created, e.g. *./bin* in QEMU top directory. Enter *./bin* and execute:

```
$ ../configure --target-list=mips-softmmu,mips-linux-user
$ make
```

The first command configures the code to build system mode and user mode for MIPS. The code is compiled using *make*. This creates system mode – *./bin/mips-softmmu/qemu-system-mips* – and user mode binaries – *./bin/mips-linux-user/qemu-mips*.

5.2 Running Linux OS and programs on MIPS in QEMU

This section assumes that Linux has been compiled using Buildroot, as described in Chapter 4. The following options have to be passed to QEMU to boot the operating system from the directory containing kernel image (*vmlinux*), root file system (*rootfs.ext2*) and optionally additional drive (*datafs.ext2*).

```
$ path_to_qemu_dir/bin/mips-softmmu/qemu-system-mips -kernel vmlinux
-hda rootfs.ext2 -hdb datafs.ext2 -append "root=/dev/hda rw panic=10"
-nographic -no-reboot
```

The extra message passed using *-append* informs the system that the root is on *hda* drive and that Linux should reboot 10 seconds after kernel panic occurs. Together with *-no-reboot* it gives a safe way to exit QEMU if some fatal error happens. As graphical interfaces are not supported, *-nographic* has to be used. Figure 5.1 shows a console with Linux running on MIPS in QEMU after executing the described command.

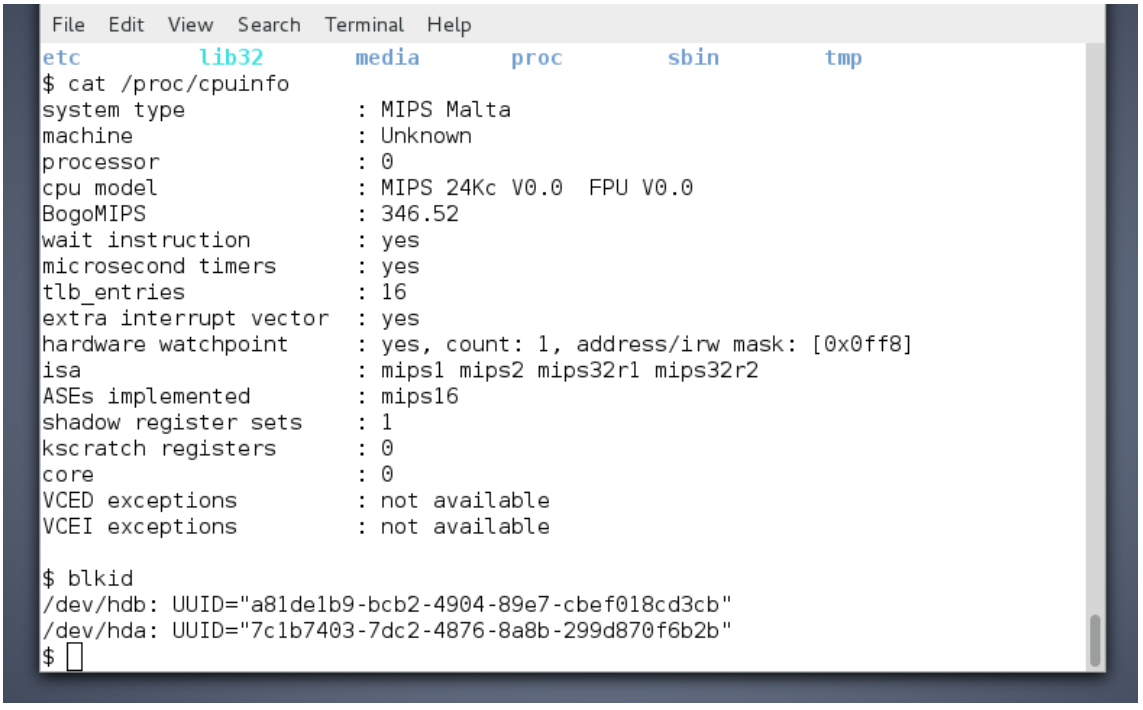


Figure 5.1: Linux running on MIPS architecture in QEMU system mode

QEMU user mode can be started with:

```
$ qemu-mips PROGRAM_NAME
```

where PROGRAM_NAME is a binary compiled for MIPS architecture. If the binary has been compiled using Buildroot toolchain, it may be necessary to include a directory containing libraries in the command:

```
$ path_to_buildroot_dir/bin/mips-linux-user/qemu-mips
-L path_to_buildroot_dir/output/target/ PROGRAM_NAME
```

It is important to specify all the options before the binary name, otherwise they are ignored.

5.3 Implemented Cache Command Line Options

This section describes all the options added to QEMU in this project. They are separated into three groups: options configuring caches, options specifying how system should deal with caches, and options used to display live cache profiling information.

5.3.1 Cache Configuration

-dcache *N*x*M*_*tp*_*rep* -icache *N*x*M*_*tp*_*rep* Examples: -dcache 1x512_dm -dcache 8x1024_2w_lfu -icache 4x1024_4w_lru -icache 2x256_2w_rnd	Specifies L1 data or L1 instruction cache configuration. The argument configures: *N* – number of 32-bit words in a cache line *M* – total number of cache lines *tp* – cache type *rep* – replacement algorithm Available cache types: 'dm' – direct-mapped, '2w' – 2-way set-associative, '4w' – 4-way set-associative. Available replacement algorithms: 'lru' – least recently used, 'lfu' – least frequently used, 'rnd' – random. For direct-mapped, cache replacement algorithm field is ignored, so *rep* can be omitted in argument string. Available sizes: *2kB:* 1x512_tp_rep 2x256_tp_rep *2kB:* 4x128_tp_rep 8x64_tp_rep *4kB:* 1x1024_tp_rep 2x512_tp_rep *4kB:* 4x256_tp_rep 8x128_tp_rep *8kB:* 1x2048_tp_rep 2x1024_tp_rep *8kB:* 4x512_tp_rep 8x256_tp_rep *16kB:* 1x4096_tp_rep 2x2048_tp_rep *16kB:* 4x1024_tp_rep 8x512_tp_rep *32kB:* 1x8192_tp_rep 2x4096_tp_rep *32kB:* 4x2048_tp_rep 8x1024_tp_rep
-l2cache *N*x*M*_*tp*_*rep* Examples: -l2cache 4x16384_dm -l2cache 16x65536_4w_lru	Specifies L2 unified cache configuration. Same as *icache* and *dcache* but different sizes are supported: *256kB:* 4x16384_tp_rep 8x8192_tp_rep *256kB:* 16x4096_tp_rep 32x2048_tp_rep *512kB:*4x32768_tp_rep 8x16384_tp_rep *512kB:* 16x8192_tp_rep 32x4096_tp_rep *1MB:* 4x65536_tp_rep 8x32768_tp_rep *1MB:* 16x16384_tp_rep 32x8192_tp_rep *2MB:* 4x131072_tp_rep 8x65536_tp_rep *2MB:* 16x32768_tp_rep 32x16384_tp_rep *4MB:* 4x262144_tp_rep 8x131072_tp_rep *4MB:* 16x65536_tp_rep 32x32768_tp_rep

Table 5.1: Cache configuration options

5.3.2 Operation Modes

-transparent_cache	It has been found that configuration of MIPS registers with actual cache information may lead to freezing Linux kernel. Using this option, hardware configuration is disabled and cache operations are treated as *nop*s. As a consequence, only actual load and store instructions affect hit/miss counters. By default, L1 caches are treated as non-transparent and L2 as transparent (off-chip). Using *transparent_cache* makes all caches transparent.
–onchip_l2	Enables non-transparent L2 cache. MIPS registers are configured with actual L2 values, cache operations are enabled for L2. This option is suppressed by *transparent_cache*.

Table 5.2: Cache operation options

5.3.3 Live Display with Gnuplot

-gnuplot_cache *string* Examples: -gnuplot_cache hmstlo -gnuplot_cache hm -gnuplot_cache shot	Enable live hit/miss count display for L1 caches in gnuplot. Gnuplot package has to be installed in the system (for more information see [18]). This option should be used together with *icache* or *dcache*. The available arguments are: 'h' - instruction cache *h*it count graph 'm' - instruction cache *m*iss count graph 's' - data cache *s*tore hit count graph 't' - data cache s*t*ore miss count graph 'l' - data cache *l*oad hit count graph 'o' - data cache l*o*ad miss count graph Options should be passed together in one string, as shown in the examples. Any character that is passed in a string which is not defined here is ignored.

Table 5.3: Gnuplot live display option

5.4 Output Logs and Post-processing

Passing -*x*cache NxM_tp_rep as an option to QEMU generates a corresponding log file for each cache configuration option used: *log-xcache-NxM_tp_rep.csv*. This file contains cache line number, hit count and miss count separated by ',' (L1 instruction cache and L2 cache). L1 data cache has separate counters for load and store instructions and, therefore, data is dumped in this order: cache line number, store hit count, store miss count, load hit count, load miss count. Generated files are standard CSV files so it is possible to use any plotting software to draw hit/miss figures. This project uses python scripts for the analysis of output log files. For cache performance use *QEMU-off-line-Analysis.py* (Section 9.4.2), and cache profiling can be performed using *QEMU-off-line-Profiling.py* (Section 10.1).

5.5 Known Problems with Linux Kernel

5.5.1 Problems Encountered During Cache Testing

Linux kernel version 3.12.1, which is used in the project, does not support all cache sizes listed in Table 5.1. When only one L1 cache is specified, kernel panic happens. Only 4 and 8 words in a L1 cache line are supported, using 1 and 2 words freezes the system on boot. Only small sizes of L2 are detected, when larger L2 is used, it is ignored by the system (treated as off-chip). In all these cases using -transparent_cache bypasses the problem but it also affects hit/miss counters as cache operations are not used then.

5.5.2 Possible Reasons

Linux supports configurations which are commonly used for MIPS processor. Only unusual values cause problems.

The problem with cache line size is caused by not assigning appropriate configuration values in kernel headers (*path_to_buildroot_directory/output/build/linux-headers-3.12.1/arch/mips/mm/c-r4k.c*), e.g.:

```
static void r4k_blast_dcache_setup(void)
{
        unsigned long dc_lsize = cpu_dcache_line_size();

        if (dc_lsize == 0)
                r4k_blast_dcache = (void *)cache_noop;
        else if (dc_lsize == 16)
                r4k_blast_dcache = blast_dcache16;
        else if (dc_lsize == 32)
                r4k_blast_dcache = blast_dcache32;
```

```
        else if (dc_lsize == 64)
                r4k_blast_dcache = blast_dcache64;
}
```

Listing 5.1: Setting up cache configuration in Linux kernel based on line size

As can be seen in Listing 5.1 only 4 and 8 words are included in the if statement. When a value is different than 0, 16, 32 or 64 bytes in a cache line, nothing is done in the function.

L2 cache is only detected if MSB of L2 cache size configuration field is 0. Otherwise, Linux ignores L2.

Chapter 6

QEMU Command Line Options

This chapter concentrates on how command line options, implemented in this project, are processed and shared between different parts of QEMU system. For the description of all the options and their usage, refer to Chapter 5.

QEMU system mode handles command line options in a different way than QEMU user mode. Section 6.3 of this chapter gives an in-depth view on how a file with mixed Texinfo data and C definitions is used to parse options and generate documentation for the system mode. The user mode, on the other hand, uses simpler solution with a structure, which declares options, located in the same file as its main function. Although option parsing is different, all the options implemented in this project are processed by the same functions in system and user mode. These functions update the fields of a *MipsCacheOpts* structure, which is used to create caches of appropriate type and size and deal with other command line options.

6.1 MipsCacheOpts Struct

MipsCacheOpts structure defines variables characterizing L1 data cache, L1 instruction cache and L2 unified cache. In addition to that, some fields are added to choose which data to use for live display or switch between different operation modes, e.g. *-onchip_L2*, that treats L2 cache as if it was on-chip, rather than off-chip. All the variables can be seen in Figure 6.1.

Structure: MipsCacheOps

```
char d_opt[20];
char i_opt[20];
char l2_opt[20];

unsigned char transparent_cache;
unsigned char onchip_l2;

unsigned int *d_way_mask;
unsigned int *i_way_mask;
unsigned int *l2_way_mask;

unsigned char use_d;
unsigned char d_replacement;
unsigned char d_way_width;
unsigned int d_ways;
unsigned int d_no_of_lines;
unsigned int d_lines_per_way;
unsigned int d_offset_width;
unsigned int d_index_width;
unsigned int d_index_mask;

unsigned char use_i;
unsigned char i_replacement;
unsigned char i_way_width;
unsigned int i_ways;
unsigned int i_no_of_lines;
unsigned int i_lines_per_way;
unsigned int i_offset_width;
unsigned int i_index_width;
unsigned int i_index_mask;

unsigned char use_l2;
unsigned char l2_replacement;
unsigned char l2_way_width;
unsigned int l2_ways;
unsigned int l2_no_of_lines;
unsigned int l2_lines_per_way;
unsigned int l2_offset_width;
unsigned int l2_index_width;
unsigned int l2_index_mask;

FILE *icache_log;
FILE *dcache_log;
FILE *l2cache_log;

uint64_t *d_ld_hit_cnt;
uint64_t *d_ld_miss_cnt;
uint64_t *d_st_hit_cnt;
uint64_t *d_st_miss_cnt;
uint64_t *i_hit_cnt;
uint64_t *i_miss_cnt;
uint64_t *l2_hit_cnt;
uint64_t *l2_miss_cnt;

uint64_t tlb_error_cnt;

gnuplot_ctrl *gp_dcache_ldhit;
gnuplot_ctrl *gp_dcache_ldmiss;
gnuplot_ctrl *gp_dcache_sthit;
gnuplot_ctrl *gp_dcache_stmiss;
gnuplot_ctrl *gp_icache_hit;
gnuplot_ctrl *gp_icache_miss;
unsigned int gnuplot_max_y;
```

Figure 6.1: Structure holding cache configuration variables

The description of the structure uses *x* instead of *d*, *i* and *l2* for field names to relate to all three caches at the same time. It also takes *-xcache 4x1024_2w_lru* as an example of the command line argument specifying cache and its configuration.

The whole configuration string (*4x1024_2w_lru*) gets stored in *x_opt* character array. It is used as a part of a name of log files. The field *use_x* is set to 1 for each *-xcache* specified in command line arguments. Other 'switch' variables *transparent_cache* and *onchip_l2* are assigned to 1 if *-transparent_cache* and *-onchip_l2* are passed respectively. Once the configuration string is processed, the following fields get updated:

- *x_offset_width* – number of bits used to address bytes in a cache line; determined by a number of words in a cache line; in this example: *4* words – 4 bit *x_offset_width*
- *x_no_of_lines* – total number of lines in a cache (all ways); *1024* read from the string – 1024 *x_no_of_lines*
- *x_index_width* – another parameter specifying number of lines but in bits; *1024* lines – 10 bits *x_index_width*
- *x_index_mask* – mask used to separate index from tag; 10 bits index width – 1111111111_2 *x_index_mask*
- *x_ways* – number of ways of a set-associative cache; *2w* – 2 *x_ways*
- *x_way_width* – number of bits used to address a way of a cache; 2-way set-associative *x* cache – 1 bit *x_way_width*
- *lines_per_way* – number of cache lines per way; 1024 lines in total and 2-way set-associative *x* cache – 512 *lines_per_way*
- *x_way_mask* – cache memory is allocated using total number of lines; this field is used to address different ways in a cache; for 2-way set-associative *x* cache

with 10 bits index width, the array contains two way masks – 0000000000_2 and 1000000000_2

- *x_replacement* – replacement algorithm used in a cache; *lru* – Least Recently Used, *x_replacement* = 2 (according to algorithm encoding, Table 6.1)

Algorithm	None	LRU	LFU	Random
Code	0	2	4	6

Table 6.1: Replacement algorithm encoding

In addition to that, data logging uses 64 bit counters for each cache line *x_hit_cnt* and *x_miss_cnt* (separate counters for load and store instructions are used in L1 data cache). The value of each counter is stored to a file (*xcache_log* file pointers). The *MipsCacheOpts* structure contains one more counter, which is incremented each time a physical address cannot be obtained from a virtual address (both are needed for a proper cache operation). Refer to Section 7.7.1 for more information.

Pointers to *gnuplot_ctrl* are used to access different instances of Gnuplot structure. This structure defines parameters required for live hit/miss count display using Gnuplot software. The field *gnuplot_max_y* is common to all of them and specifies the y-range of the graphs.

Some of the options described in this section may seem to be unnecessary duplications of others, e.g. *x_no_of_lines* is the same as $2^{x\_index\_width}$ and *x_index_mask* is the same as *x_no_of_lines* - 1. The reason for storing all these values is that they get used very often, even every instruction. As a result of storing them, they do not have to be re-calculated each time.

6.2 Code Flow

When a user executes the system mode or the user mode, functions from different files are used to process command line arguments. Figure 6.2 gives an overview on the code flow between the files and explains what happens at each level without going into the implementation details. The file paths take QEMU top level folder as the current directory.

The *MipsCacheOpts* structure, described in Section 6.1, is declared in *./target-mips/mips-cache-opts.c* together with functions updating its fields. They are shared between files by *./target-mips/mips-cache-opts.h* header file.

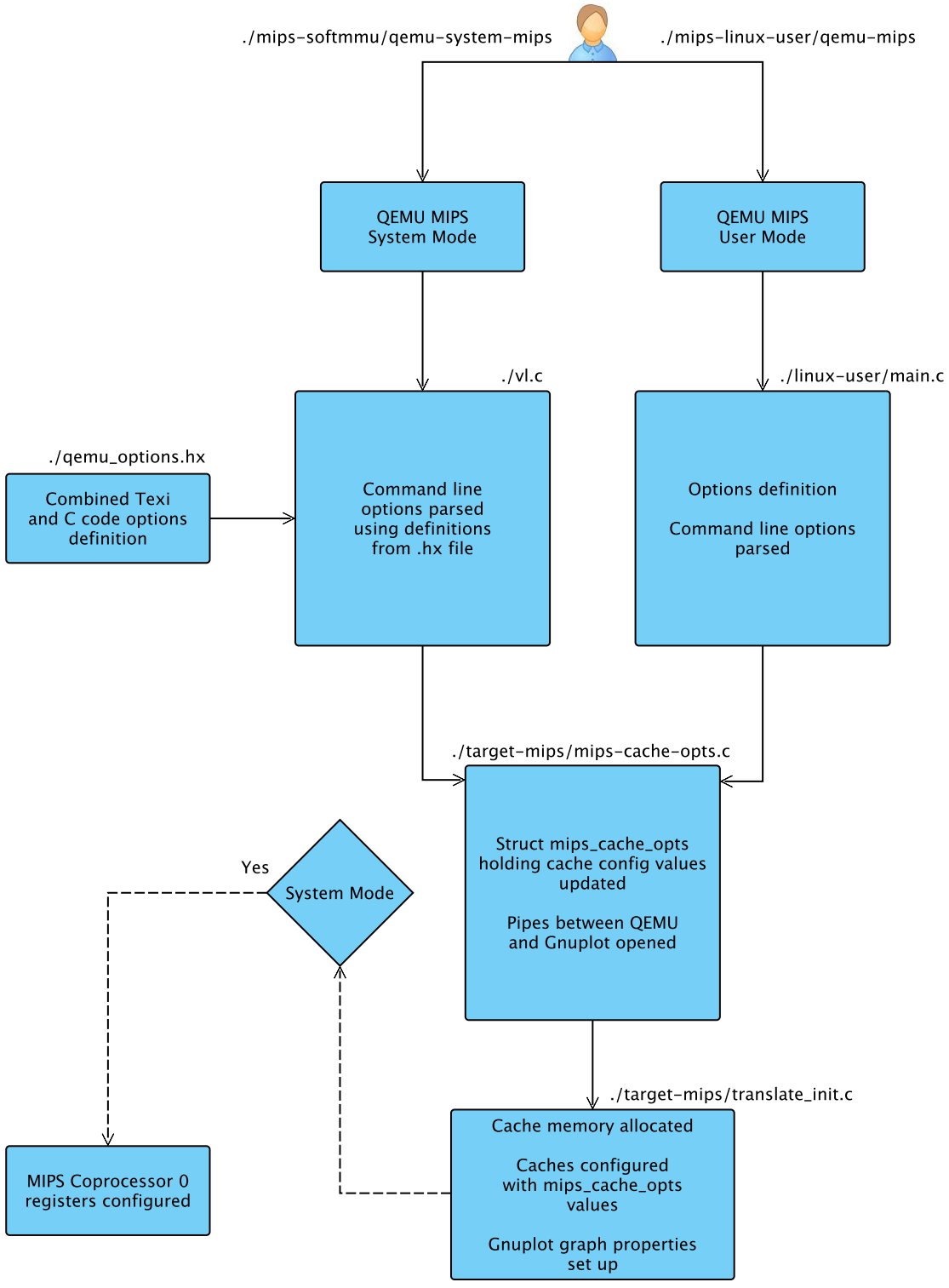


Figure 6.2: Cache configuration with command line options

6.3 QEMU System Mode Options Parsing

As shown in Figure 6.2, the *./qemu_options.hx* is a source file for both C definitions and documentation. When QEMU is compiled, all the command line arguments are read from this file and the appropriate label is given to each of these arguments. This is an example how to define a new option:

```
DEF("new_option", has_arguments?, option_label,
  "-new_option option_arguments (usage example)"
  "Short description for command line manual",
  architecure specification)
STEXI % Documentation starts here
@item -new_option option_arguments (usage example)
@findex -new_option
Long description for html documentation.
ETEXI % Documentation ends here
```

The text in *DEF()* is used to generate C code for QEMU. It allows to specify an option name (*new_option*), whether arguments are necessary (either *0* or *HAS_ARG*), option label (usually *QEMU_OPTION_xxxx*), short description of usage, and supported architecture (e.g. *QEMU_ARCH_MIPS*). The lines between *STEXI* and *ETEXI* are used to create Texinfo file – *qemu-options.texi* in QEMU build folder. The documentation is generated in HTML file. However, as a Texinfo file is used, it is also possible to produce PDF, DVI and other formats [5].

The QEMU system mode *main* function, declared in *./vl.c* file, parses command line options. Firstly, it checks if the option name is known, and by using the second field of DEF(), it determines whether an argument is required for this option. After that, based on option label, it calls the appropriate function using a case statement. The list of command line options implemented in this project, their labels and corresponding functions can be seen in Table 6.2. The functions presented in the table, are described in Section 6.5, which gives more information about their code.

Option	Label	Function processing this option
-icache	QEMU_OPTION_icache	`proc_mips_cache_opt ('i',optarg)`
-dcache	QEMU_OPTION_dcache	`proc_mips_cache_opt ('d',optarg)`
-l2cache	QEMU_OPTION_l2cache	`proc_mips_cache_opt ('u',optarg)`
-transparent_cache	QEMU_OPTION_transparent_cache	`set_transparent_cache ()`
-onchip_l2	QEMU_OPTION_onchip_l2	`enable_onchip_l2 ()`
-gnuplot_cache	QEMU_OPTION_gnuplot_cache	`gnuplot_create (optarg[gpl_i])`

Table 6.2: QEMU system mode options, their labels and functions

6.4 QEMU User Mode Options Parsing

The User Mode takes a simpler approach to option handling. It defines a structure for each option in *./linux-user/main.c* file:

```
struct qemu_argument {
    const char *argv;
    const char *env;
    bool has_arg;
    void (*handle_opt)(const char *arg);
    const char *example;
    const char *help;
};
```

Listing 6.1: QEMU User Mode options definition struct

It contains: option string (**argv*), a label (**env*), variable set if arguments are required (*has_arg*), pointer to a function processing the option (**handle_opt*), string showing an example usage (**example*) and string describing the option (**help*).

Four options are added in this project:

```
{"dcache","QEMU_CACHE_D",true, handle_arg_dcache,
 "nxm_tp_rep","choose d-cache size, type and repl. algorithm"},
{"icache","QEMU_CACHE_I",true, handle_arg_icache,
 "nxm_tp_rep","choose i-cache size, type and repl. algorithm"},
{"l2cache","QEMU_CACHE_L2",true, handle_arg_l2cache,
 "nxm_tp_rep","choose L2 cache size, type and repl. algorithm"},
{"gnuplot_cache","QEMU_CACHE_PLOT",true, handle_arg_gnuplot,
 "","plot live hit/miss graphs"},
```

Listing 6.2: User Mode options added

The other options described in system mode section are not needed here. This is because they are responsible for updating MIPS registers, and that is not used when running the user mode. As can be seen in Listing 6.2, there are four functions handling cache arguments. In fact, they do not do anything else apart from calling *proc_mips_cache_opt(...)* with appropriate arguments for each cache or *gnuplot_create(...)* with a string specifying the desired plots (Table 6.2).

6.5 Processing Cache Configuration

Previous sections mentioned the *proc_mips_cache_opt(...)* function, but none of them described in details what it actually does. This function is declared in *./target-mips/mips-cache-opts.c* source file and is responsible for updating MipsCacheOpts structure fields. In fact, the file contains more functions, all of which can be seen in Figure 6.3.

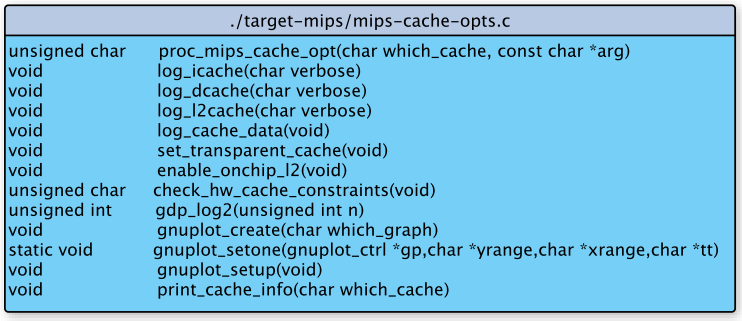


Figure 6.3: Functions declared in *./target-mips/mips-cache-opts.c*

Some of them are pretty straightforward and do the following:

- `log_`*x*`cache` – dumps *x* hit/miss counters to a file, if 1 is passed as an argument, prints log information.
- `log_cache_data` – calls `log_`*x*`cache` for each cache used, frees memory allocated for cache storage, closes opened Gnuplot pipes.

- `set_transparent_cache` – sets *transparent_cache* field in *MipsCacheOpts* structure to 1.
- `enable_onchip_l2` – sets *onchip_l2* field in *MipsCacheOpts* structure to 1.
- `gdp_log2` – returns log_2 of the input argument, if the argument is 0, return 0.
- `gnuplot_setone` – sets properties of one graph, used only by *gnuplot_setup()*.
- `gnuplot_setup` – configures selected Gnuplot plots, all properties are hard coded apart from y-range, which is determined by *gnuplot_max_y* field in *MipsCacheOpts* structure.
- `print_cache_info` – prints information about a cache, selected by the argument passed, in a user friendly format.
- `gnuplot_create`, `check_hw_cache_constraints` and `proc_mips_cache_opt` are more complicated. As it is important to understand how they work for anyone who intends to implement any other flavours of cache memory in the existing system, separate subsections have been dedicated to these functions.

6.5.1 Function `proc_mips_cache_opt(...)`

Figure 6.4 shows all the steps performed in the function. The configuration of caches works in the same way for all caches. As a result, there is no gain in creating separate functions. Instead, a set of pointers corresponding to each cache field is declared. Based on a value of the *which_cache* argument, these pointers are set to memory addresses of the fields of the chosen cache. The main advantage of this approach is simplicity – there is no need to switch between *i*, *d* and *l2* fields further in the code.

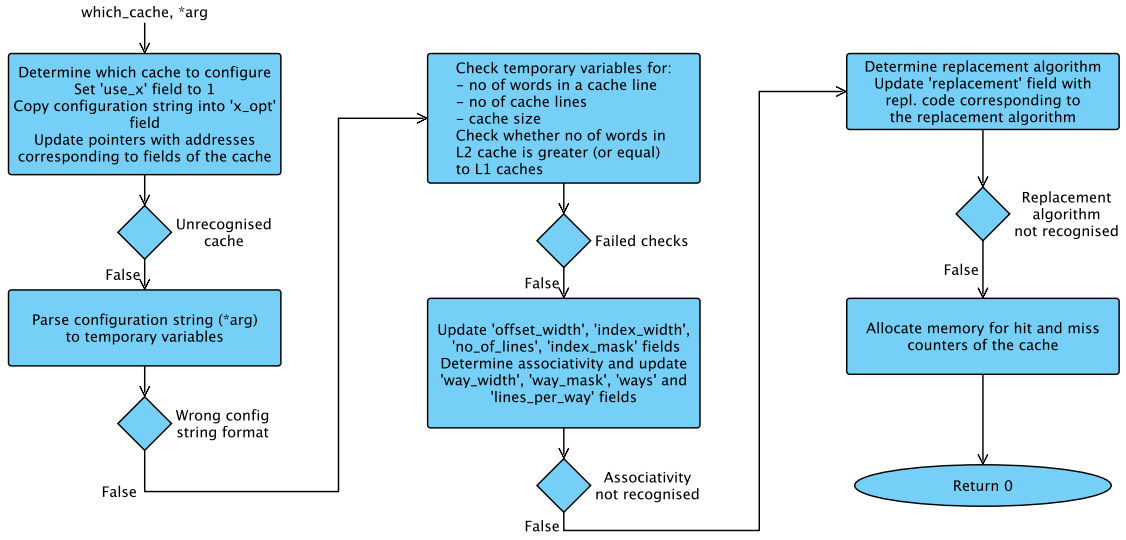


Figure 6.4: Stages of the `proc_mips_cache_opt(...)` function

The other argument of the function – **arg* – is the configuration string already described in Chapter 5 (e.g. *1x1024_4w_lru*). The string is parsed using *sscanf()* with the following format: `"%ux%u_%2s_%3s"`, where the first value corresponds to the number of words in a cache line and the second one to the number of cache lines. The two character string determines cache associativity, and the three character one, the replacement algorithm. All of them are stored in temporary variables. Various checks are performed at this stage to determine whether the cache meets the required specification. Table 2.3 shows the cache configurations used in this project. Whenever there is a need to add different cache type, size, another replacement algorithm, etc., this part of the code needs to be updated with this additional configuration. Once all the checks have passed successfully and all the fields have been updated, the function returns 0. Otherwise, an error is signalled with 1. Figure 6.4 does not show the error signals just to make the diagram more clear.

6.5.2 Function `check_hw_cache_constraints(...)`

Using this function it is possible to exclude any configurations, which are not supported by the emulated hardware. For instance, in MIPS processor registers the number of cache lines per way must be a multiple of 64 (Subsection 6.7). Some available options have less lines, e.g. *-xcache 8x64_4w_xxx, -xcache 4x128_4w_xxx*. These are checked in this function and error is returned if they are used.

Any modifications to the code should put all the constraints in this function. It returns 0 on success and 1 if some conditions are not met.

6.5.3 Function `gnuplot_create(...)`

A C module is used for communication between QEMU and Gnuplot [9]. It has been simplified and optimized for the purpose of this project. All Gnuplot functions are declared in *./target-mips/gnuplot_i.h* header file and defined in *./target-mips/gnuplot_i.c* source file (QEMU directory). The details are not provided here but these files are responsible for the following actions:

- Creating a pipe between Gnuplot and QEMU – *gnuplot_init(...)*
- Sending commands to Gnuplot to configure plot display – *gnuplot_cmd(...)*
- Creating temporary files and plotting them – *gnuplot_plot_x(...)*
- Closing the opened pipe – *gnuplot_close(...)*

The *gnuplot_create(char which_graph)* function decodes the character passed as an argument and chooses appropriate gnuplot structure pointer from *MipsCacheOpts*. It

calls *gnuplot_init(...)* function from *gnuplot_i.h* and assigns the returned address to this pointer. The pointer is used later to set up graph properties and plot data.

It is important to note that *gnuplot_create(...)* can only create one pipe per call. As a consequence, it has to be called in a loop with the argument string passed with *gnuplot_cache* option. The main reason for such implementation is its simplicity. A switch statement, used in the function, provides an efficient way of filtering wrong or repeated characters. As a result, any arbitrary string can be used and only non-repeated valid character codes are used. Others are simply ignored. Table 6.3 shows which pointer in *MipsCacheOpts* is used for which character.

Character code	*MipsCacheOpts* pointer
'h'	*gp_icache_hit
'm'	*gp_icache_miss
's'	*gp_dcache_sthit
't'	*gp_dcache_stmiss
'l'	*gp_dcache_ldhit
'o'	*gp_dcache_ldmiss

Table 6.3: *gnuplot_create(...)* character codes associated with *MipsCacheOpts* pointers

6.6 Cache Memory Allocation

The memory for caches is allocated using *cache_init(...)* function added in *./target-mips/translate_init.c* file . Two of the *MipsCacheOpts* fields are read: *use_x* and *x_no_of_lines*. They are used to determine which caches are present and their size. Refer to Section 7.4.2 for more information about cache storage.

It is important to mention *g_malloc0(...)*, which is used instead of *calloc(...)* for dynamic memory allocation and initialisation with 0. QEMU developers do not allow the latter one as it is less efficient, does not terminate on allocation failure and does not provide any checks for memory leakage. [4] [2]

6.7 Updating MIPS Coprocessor 0 Registers

MIPS uses a set of *coprocessors* to provide additional functions, which are not included in the processor. For instance, Coprocessor 1 is responsible for hardware floating-point

operations. All coprocessors (apart from Coprocessor 0) are optional and only included if required in a specific application. Coprocessor 0, on the other hand, plays an indispensable role in the MIPS system and is sometimes called the *System Control Coprocessor*. It performs many functions – CPU configuration, cache configuration and control, interrupts and MMU control, etc. – without which MIPS cannot be used. [16, pp. 53-55]

6.7.1 Coprocessor 0 Config*X* Registers

This project concentrates on caches and, therefore, cache control using cache operations and cache configuration in Coprocessor 0 registers are the most important here. Cache operations are described in Section 2.4 and their implementation in Section 7.5.2. This part examines cache configuration in Coprocessor 0.

Coprocessor 0 contains a set of *Config* registers, which are used mostly to pass the processor configuration to an operating system. Figure 6.5 shows *Config1* and *Config2* registers, which are used to specify L1, L2 and L3 cache properties.

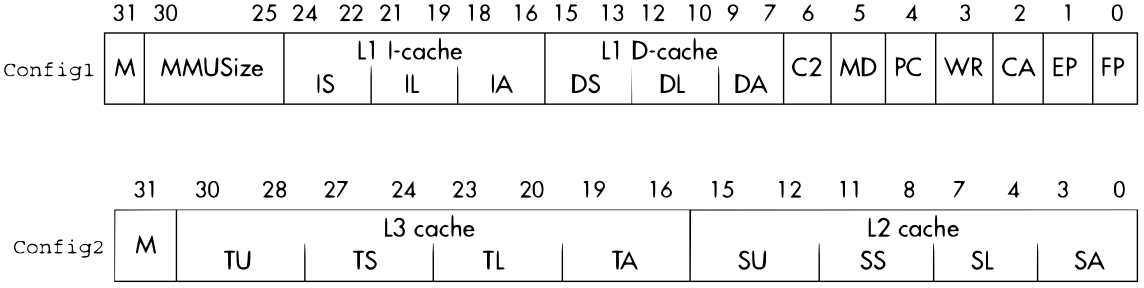


Figure 6.5: Config1 and Config2 Registers (sourced from [16, p. 71])

Each cache field consists of 3 subfields – 3-bits each in L1, 4-bits each in L2 and L3. They are used to set:

- The number of cache lines per way: 64 x 2^S
- Cache line size in bytes: 2 x 2^L
- Associativity: (A+1)-way set-associative [16, pp. 71-72]

6.7.2 Modifying CP0 Register Values in QEMU

The registers are only configured when the QEMU system mode is used. The list of supported MIPS CPUs and their configuration is stored in *mips_def_t* structure in *./target-mips/translate_init.c* file. This structure contains *CP0_Config1* and *CP0_Config2* fields, which correspond to *Config1* and *Config2* registers. Before QEMU starts processing instructions, it needs to determine the MIPS processor model and its configuration. It does

that by passing a string, specifying the processor model, to *cpu_mips_find_by_name(...)* and hence it checks if the model is supported. If so, it returns the CPU configuration.

To update the *mips_def_t* structure with the values from *MipsCacheOpts* before the configuration gets read by the system, the *config_cache()* function is added and executed at the beginning of *cpu_mips_find_by_name(...)*. This function clears the default values of S, L and A subfields in Config1 and Config2 and uses *MipsCacheOpts* fields to set up the registers with actual cache sizes. It also executes *check_hw_cache_constraints(...)* to make sure the configuration is supported by hardware. If for some reason an operating system does not work with some unusual values in the registers, *config_cache()* can be disabled with *-transparent_cache* option for all caches. By default, the function does not configure L2 cache and clears corresponding register values. To update L2 register subfields with actual L2 values, *-onchip_l2* option should be used. Listing 6.3 shows how new values of S, L and A subfields are calculated for L1 instruction cache. Listing 6.4 contains a part of the boot information from the Linux kernel running in QEMU and configured with: `-dcache 4x256_2w_lru -icache 8x512_4w_lfu -l2cache 32x2048_4w_rnd -onchip_l2`. It confirms that the values are properly set up in registers and read by the operating system.

```
is = (mips_cache_opts.use_i)? (mips_cache_opts.i_index_width -
    6 - mips_cache_opts.i_way_width) : 0;
il = (mips_cache_opts.use_i)? (mips_cache_opts.i_offset_width - 1) : 0;
ia = (mips_cache_opts.i_way_width)? ((1 << mips_cache_opts.i_way_width)
        - 1) : 0;
```

Listing 6.3: Calculation of new values for L1 instruction cache subfields in Config1

```
Primary instruction cache 16kB, VIPT, 4-way, linesize 32 bytes.
Primary data cache 4kB, 2-way, VIPT, no aliases, linesize 16 bytes
MIPS secondary cache 256kB, 4-way, linesize 128 bytes.
```

Listing 6.4: Cache information read by Linux and displayed during boot

Chapter 7

System Implementation

A full description of the implemented design is presented here, which will build on the previous sections that gave an overview of how the system may be used. As there are several parts involved, it may be useful to refer to Figure 7.1 for an idea of how the various parts interact.

7.1 Design considerations

Before implementation began, the various aspects of the project were analysed, and decisions were made on how to design them best. In several cases it was required to converse further with the customer in order to make these decisions correctly.

7.1.1 Cache Store

One of the first questions to be addressed was that of how 'real' the cache emulation needed to be. On one hand, it would be possible to actually store data in the cache, act as a real cache would and actually provide the data when it is required, and thus work around QEMU's built-in memory management. Alternatively, the cache could be purely for analysis, not store any actual data, and continue to rely on QEMU for memory management.

After investigating the feasibility of both options, and discussing this with the customer, it was decided that actually storing data would be more effort than it was worth. In particular, it would mean more memory accesses would be required, and it would create more opportunities for bugs to occur. QEMU already has a complex and mature memory management system, and thus it was deemed unnecessary to place another layer above it.

A second aspect to this is the question of what data and statistics should be stored in the cache, if the actual data is not. It is clear that at the very minimum, to perform hit rate analysis, a count of hits and misses per cache line is required. Several other parameters were investigated as well, such as storing the list of instructions run, or storing the process ID that caused a hit or miss. However, after investigating QEMU more, and discovering 'user mode', it was decided that this data could be made partly redundant by user mode. If only one particular process wants to be examined, then it is far simpler to run it in user mode than to run in system mode and only dump its hit/miss data. Either way will reveal where the program is inefficiently coded.

7.1.2 Data Reporting

One of the aims in this project is to build a system that can be used to perform a variety of analyses, from cache design, to application profiling. Thus substantial thought went into the best methods to perform these tasks, as well as what data would be required to perform them. Section 9.1 goes into more detail about what to do with the data once acquired.

In terms of data reporting, there were several considerations. Firstly, it was unclear initially when precisely the cache statistics would be given to the analysis tools. One option was to gather the hit/miss statistics and then dump them all at the end of program execution, and another possibility was to incrementally dump the data while the program ran. The first option is simpler, but the second presented the potential of implementing some form of live data analysis, although it was not clear how useful this would be. It was decided to initially focus on developing analysis tools that could deal with a single dump of data at the end of execution, and that if there was time live analysis would be investigated. This turned out to be a good choice, as the live analysis implemented was a relatively simple addition to the system already in place.

Another data reporting consideration was how to dump the data – ASCII or binary, and, what data layout, were the main options debated. However, as this is a fairly simple part of the system, and since empirical data always helps, both ASCII and binary logging routines were implemented. It was found that although the binary logs greatly reduced file size, they also increased parsing difficulties, as a non standard format was employed. ASCII dumps, on the other hand, were found to be quite flexible, especially when the data format was CSV (comma separated values), for example, which allowed the files to be viewed and manipulated in standard spread-sheeting applications. It was this reason that made ASCII dumps the better choice in this project, and thus they are used in the final version of the system.

7.1.3 Other Considerations

It was agreed at the start that the system should be as flexible as possible, and thus, hopefully, more useful. Although the system designed targets specifically, several design choices about the structure were made in order to make it very simple to add similar cache emulation to other platforms. The very first design decision made was about how to integrate the cache system into QEMU. After discussing the problem with the customer, and investigating QEMU's internal structure, it was decided that using helpers would be the best way to do it.

The cache designs and replacement strategies implemented may seem fairly arbitrary, but they were discussed in the group and with the customer. LRU, LFU, and Random were specific replacement strategies that were required to be implemented, although other types do exist. This however, is left for future work.

7.2 Overall System Design

Several files were added to the QEMU source tree. A list of these files, along with a short description of their purpose is given here so that the following sections are seen in better context. In addition, a full list of the files changed and added is provided in Appendix C, and may be useful for reference.

`target_mips/cache_helper.c`
- Implements the MIPS specific helpers required to hook into program execution.

`target_mips/cache.h`
- Contains generic cache definitions, structures, function prototypes etc.

`target_mips/cache.c`
- Implements basic direct mapped cache functionality.

`target_mips/replacement_{lfu,lru,rnd}.c`
- These three files implement LFU, LRU and Random cache replacement algorithms respectively.

`target_mips/mips-cache-opts.h`
- Contains the cache options structure used to control user interaction.

`target_mips/mips-cache-opts.c`
- Contains all user interaction code (parsing command line arguments, logging, etc).

In addition, the following files were modified to integrate the cache system in QEMU.

- target_mips/cpu.h

- target_mips/helper.h
- target_mips/translate.c
- target_mips/translate_init.c
- target_mips/helper.c

A high level block diagram is given in Figure 7.1, which shows the main parts of the implemented system. These parts are all designed to be fairly modular, in order to facilitate testing and debugging, and as such they may also be explained individually fairly easily.

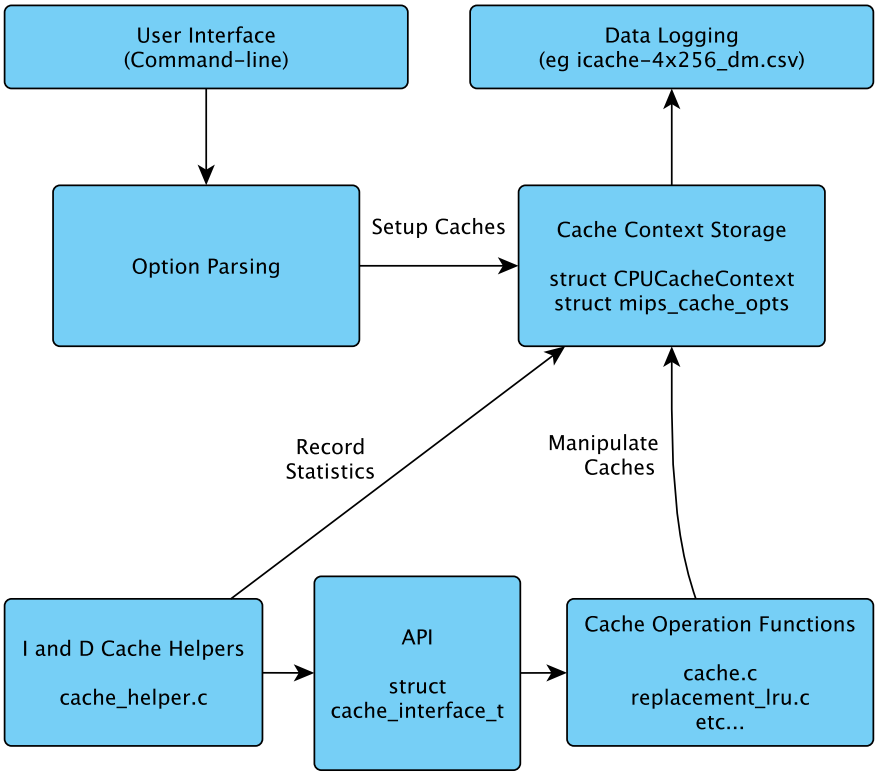


Figure 7.1: High level block diagram of system implemented in QEMU

7.3 API

From the outset it was known that the system was required to be flexible in order to facilitate analysis of cache designs. One of the obvious requirements for this is that it should be simple to allow a system to have any possible combination of L1 and L2 cache types and sizes. The size of each cache may be obtained at run time and used to initialise sufficient memory, and hence this parameter is easily controlled. However, each cache will require different functions to be called to operate on the data, according to whichever type of cache it is configured as (direct mapped, LRU, etc). An API/interface has been created, which is easily initialised at run time, and allows the calling functions

to be independent from the cache type. Thus, the helpers described in Section 7.5 do not need to know anything about the underlying cache type, as the function calls go through this interface.

The API is comprised of one struct, *cache_interface_t*, which is defined in cache.h, shown in Listing 7.1, and is essentially a list of function pointers. This structure (currently) contains five function pointers – the main lookup function that is called to access the cache, and four other functions that perform various cache operations. However, it may be easily extended to include more functions as the system is built upon and requires greater functionality. The purpose of all these functions will be explained in subsequent sections.

```
struct cache_interface_t {
  /**
   * lookup: The main cache lookup function.
   */
  int (*lookup)(cache_item_t *cache, uint32_t index, uint32_t tag,
                unsigned int *mask, uint8_t n_indexes, int *idx_used);
  /**
   * invalidate: mark a cache line as invalid and remove lock
   */
  void (*invalidate)(cache_item_t *cache, uint32_t index, uint32_t tag);
  /**
   * hit_invalidate: mark a line as invalid IF it is currently valid
   */
  void (*hit_invalidate)(cache_item_t *cache, uint32_t index, uint32_tag,
                         unsigned int *mask, uint8_t n_indexes);
  /**
   * fill_line: fill a specified cache line (mark it as valid etc)
   */
  int (*fill_line)(cache_item_t *cache, uint32_t index, uint32_t tag,
                   unsigned int *mask, uint8_t n_indexes, int *idx_used);
  /**
   * fetch_lock: fill a cache line AND mark it as locked
   */
  void (*fetch_lock)(cache_item_t *cache, uint32_t index, uint32_t tag,
                     unsigned int *mask, uint8_t n_indexes);
};
```

Listing 7.1: Cache interface (API) structure

With the above structure it becomes relatively easy to create a different set of functions for each cache type, and, at initialisation time, select the correct set and make it accessible to the cache helpers.

As four different replacement algorithms (and thus, sets of functions) are implemented in this system, it was decided to split the code up into separate files – one for each algorithm. In addition to the sets of functions, an instance of *cache_interface_t* was created for each

replacement algorithm, and initialised to point to the correct functions. However, it was realised that in some cases it was not necessary to define new functions, as there is some similarity between different cache types. For example, none of the set associative caches needed to do anything special for a Hit Invalidate instruction, and the simple direct mapped Hit Invalidate function was sufficient. To prevent code duplication, the direct mapped functions were defined in cache.h and cache.c, thus making them accessible in all the other replacement algorithm files.

With the separate function tables available, a *cache_interface_t* pointer may be defined to point at one of these tables, such that a TCG helper may call them through the interface. For example, the following snippet of code illustrates how this could be done, assuming that *interface_lru* is the LRU function table defined previously, and *cache_api* is the generic interface. In the first line, *cache_api* is initialised to point to the the LRU interface, and in the second line *lookup_lru* is actually being called.

```
interface_lru.lookup = lookup_lru;
cache_interface_t *cache_api = &interface_lru;
result = (*cache_api->lookup)(...);
```

7.4 Data Structures

There are several fairly important data structures used in the low level cache system; all of these will be explored in this section.

7.4.1 CPUCacheContext

Perhaps the most important structure in the system is the CPUCacheContext struct, defined in cache.h and shown in Listing 7.2. This structure holds all the required parameters for the cache to be initialised and operate, and is intended to be included in a target's CPU state structure, CPUMIPSState in this case. One of the two modifications made to *cpu.h*, for this project, was adding the cache context struct to CPUMIPSState (the other is mentioned in Section 7.5). This is done in order to make the cache context easily accessible from every part of the target that has access to the CPU context (such as a helper).

```
struct CPUCacheContext {
  cache_item_t *icache;
  cache_item_t *dcache;
  cache_item_t *l2cache;
  struct MipsCacheOpts *opts;
  cache_interface_t *icache_api;
  cache_interface_t *dcache_api;
  cache_interface_t *l2cache_api;
```

```
};
```

Listing 7.2: Primary cache control structure

The CPUCacheContext struct contains pointers to memory blocks for all three possible caches (arrays of *cache_item_t*), a pointer to the cache options parsed from the command line, and pointers to interfaces that will be used to call the correct functions for each cache type.

The entire structure is initialised in translate_init.c before any program execution takes place (this is also where the MIPS TLB is initialised). At this stage, the command line arguments have been fully parsed, and the required cache sizes and types are stored in the *mips_cache_opts* struct defined in mips-cache-opts.h. Based on these options, this structure is initialised such that the cache memory blocks are the correct sizes (depending on the number of cache lines), and the interfaces are assigned according to the cache type. This allows, for example, the I-cache helper to call the appropriate cache lookup function by using *CPUCacheContext-¿icache_api-¿lookup*.

7.4.2 Cache Storage

As mentioned previously, one design consideration was that the cache needed to be flexibly sized, and would be a 'virtual cache' – no memory would be actually stored; only the relevant tags and flag bits required for every line (see Section 2.6). This means that the storage is not dependent on the number of items per cache line – only on the number of lines in the cache, which simplifies the memory management process. The data for each line of the cache is stored in a packed structure defined in cache.h, show in Listing 7.3.

```
struct cache_item_t {
  uint_fast32_t tag          : 32;
  uint_fast32_t r_field      : 30;
  uint_fast8_t valid         : 1;
  uint_fast8_t lock          : 1;
};
```

Listing 7.3: Cache line data structure

This was designed to take up as little memory as possible, as there would be many of these structures instantiated. The first 32 bit value, *tag*, is simply the tag of the memory currently in that cache line. The 30 bit *r_field* is used by any replacement algorithms that require flags or counters for each line in order to perform their replacement strategy. The last two bits of the structure are the valid and lock flags that are required in a MIPS cache.

In translate_init.c, a new function was created, *cache_init* which, in addition to performing other tasks, allocates cache memory based on the number of lines in each cache. This is a simple process of multiplying the size of *cache_item_t* with the number of required lines, and allocating the memory using *g_malloc0*. The resulting pointer is then saved in *CPUCacheContext* as described in Section 7.4.1. One thing to note when using *g_malloc0*, instead of plain *calloc*, is that if there is not enough memory left to allocate, the program will abort. This is preferable to allowing execution to continue and then getting a segmentation fault or wrong results.

7.5 Cache Helpers

In order to model a cache in QEMU, some way of monitoring program execution is required, so that the algorithm may know when a certain part of the cache must be accessed. As was described in Chapter 3, QEMU uses instruction translation to execute guest code, and custom code (the form of helpers) may be inserted arbitrarily in this process. Helpers are used in this project to call relevant cache manipulation code when required.

There are two main categories of helpers that are added to QEMU by this project, and these may be broken down for specific functionality:

- Cache access helpers
  - D-Cache (load/store) instruction helpers
  - I-Cache instruction helper
- Cache instruction helpers
  - `cache_invalidate`
  - `cache_load_tag`
  - `cache_store_tag`
  - `cache_hit_invalidate`
  - `cache_fill`
  - `cache_fetch_lock`

Each helper is defined in cache_helper.c, and declared in helper.h using the macros provided by QEMU.

7.5.1 Access Helpers

The cache access helpers were among the first parts of the project implemented, as they form the basis of the entire system. They are inserted into translation blocks during QEMU code generation, and perform the highest cache lookup level tasks. There are two helpers in this category – one for the data cache, and one for the instruction cache. At

this level, their task is simply to call a cache lookup function, and then take appropriate action depending on whether the value returned indicated a hit or a miss. Both helpers first attempt a lookup in the L1 cache and fall-back to the L2 cache if there is a miss in the L1, and then record the hit or miss in a counter (according to the steps described in Section 2.6).

Deciding where to place the data cache helpers in the translation process was simple, as they only need to be called for Load and Store operations, and hence were placed in *gen_ld* and *gen_st* (in translate.c). Thus every Load and Store translation block also includes the data cache access code. However, the instruction cache was expected to be trickier, as it needs to be placed once in every instruction translation block. This is actually accomplished fairly easily by placing it in the *decode_opc* function, which is called once for every opcode that a translation block is generated for. Adding the helper code to every instruction clearly will have a negative impact on performance, but is a requirement of the system.

7.5.1.1 Physical vs Virtual Addresses

It was noted in the earlier section on MIPS caches (Section 2.5) that in MIPS, the L1 cache is virtually indexed and physically tagged. In order to implement this, an extra function, *get_phys_addr_cache*, was added to *target_mips/helper.c* which takes a virtual address as a parameter and returns its physical address. This also required modifying *cpu.h* to add a function prototype so that this function could be called from the cache helper function. However, in QEMU, it is only possible to retrieve the physical address when in system mode. When in user mode, a special form of address translation is used in QEMU which makes host addresses effectively transparent to the guest, and the MIPS TLB is disabled. Thus, the physical address is only used to index the cache when running in system mode.

7.5.2 Instruction Helpers

The cache in a real MIPS CPU is not 'transparent' – the CPU knows about the cache, and several instructions exist that allow the CPU to manipulate it (see Section 2.4). In the original QEMU MIPS target, these instructions were simply ignored (treated as NOPs), as there was no cache for them to operate on. These instructions have now been implemented, and perform the required actions on the cache, thus giving the OS better control of the system.

A helper is created for each of the cache operations that is supported, and a function (*gen_cache_op*) is implemented in translate.c which examines the opcode and calls the appropriate helper. Currently only the L1 cache instruction helpers are ever called, as

the L2 operations are optional according to the MIPS Specification [16]. However, the helpers are defined in a way that it is very simple to add support for L2, by writing one line of code that calls a C preprocessor macro. As the helpers only vary a small amount between each cache, macros are used to generate the helper code. Despite the fact that macros can be tricky to debug, using them results in a cleaner implementation, and less space for errors as there is less code duplication. In addition, these helpers are all fairly simple, as they typically only need to perform one action – calling the relevant cache manipulation function via the API explained in Section 7.3.

7.6 Cache Algorithms Implementation

Two main cache types were implemented in the system – direct mapped, and set associative. The set associative caches support either 2 or 4 sets, and use one of the following replacement algorithms:

- Least Recently Used (LRU)
- Least Frequently Used (LFU)
- Random

As the methods of modelling these caches algorithmically has already been explored (Section 2.6), this section will just explain how these algorithms have been integrated into the QEMU architecture.

The cache manipulation functions are all designed to be modular and completely decoupled from the rest of the system. This allowed them to be readily tested and developed without having to be in the QEMU system at all times. In addition, they were designed such that one function could be used to manipulate any cache. Each cache manipulation function takes, as a first parameter, a pointer to an array of *cache_item_t* structs which is taken as the cache to operate on. Then, depending on the function, an index and tag are passed in – these are both calculated before calling the function, so that the function does not need to perform this task. The only time the function needs to know the cache type is when dealing with a set associative cache, as the function needs to know how many indexes to use. This parameter is therefore passed to the functions that need it. As an example, the generic cache lookup function is shown in Listing 7.4, along with its documentation.

```
/**
 * lookup: The main cache lookup function.
 * @param  *cache    Pointer to the cache to operate on
 * @param  index     Index of memory access
 * @param  tag       Tag of memory accessed
 * @param  *mask     Index mask (for set-associative caches)
 * @param  n_indexes Number of indexes (sets)
```

```
 * @param *idx_used  In S.A. caches, this is set to the line that was
    used
 * @return           +ve: hit, -1: miss
 */
int (*lookup)(cache_item_t *cache, uint32_t index, uint32_t tag,
              unsigned int *mask, uint8_t n_indexes, int *idx_used);
```

Listing 7.4: Generic cache lookup function and its documentation

This is the function signature shared by all the lookup functions (including direct mapped, although it does not use the last two parameters). The *mask* parameter is an array of index masks, used by set-associative cache functions to calculate which lines to check. It is calculated during the command-line argument parsing stage, to reduce the number of operations that need to be performed in each cache lookup, and hence improve performance. The final argument, *idx_used* is set to the index of cache line that was chosen by the replacement algorithm, as this is sometimes useful to know.

The Direct Mapped cache was implemented in *cache.c* both because it was the first type to be implemented, and because it is the simplest. There is hardly any logic associated with this cache, and it was trivial to integrate into QEMU because of this.

7.6.1 Set Associative Cache Implementation

Least Recently Used and Least Frequently Used both require some form of 'memory' of the cache state, so they can correctly choose the cache index according to their respective algorithms. The *r_field* bit-field in *cache_item_t* (see Section 7.4.2) was created to provide this memory. This allows, for example, the LFU lookup function to know how frequently a certain cache line has been accessed recently, by using *r_field* as a counter. LRU uses *r_field* to rank the possible cache lines in order of how recently they were used.

Although the replacement algorithms are each given their own file in which they are defined, some functionality is shared between them. All three replacement algorithms use the 'simple' version of the cache Invalidate operation (defined in *cache.c*), and all three require the same Hit Invalidate operation. Therefore, *assoc_hit_invalidate* was also created in *cache.c*, where it can be accessed from the other replacement algorithm files. The API/interface makes this sharing process very simple, as it is essentially just assigning different function pointers to the same function.

The Random replacement algorithm was implemented only slightly differently from LRU and LFU. The primary difference is that the random number generator requires initialisation, which is done in the *cache_init* function in *translate_init.c*. The standard library function *rand()* is used to generate the numbers, and it is seeded by the current system time, which guarantees sufficient randomness for this purpose.

7.7 Problems Encountered

One known limitation of the system is that it is currently unable to identify which process caused a particular hit or miss (only applicable when running in system mode). This information could potentially be useful when doing cache profiling, although it is possible to get essentially the same information by profiling in user mode. Some time was spent investigating whether the TLB ASID (address space identifier) could be used to identify the process, but this was unsuccessful as the ASID is assigned randomly. It may be possible to acquire this information in other ways, but it was deemed to be unnecessary in the scope of this project, and left for future work.

7.7.1 Known Remaining Bugs

There is an issues with the way addresses are translated from virtual to physical in the L1 D-cache helper, when in system mode, which appears to be caused by unexpected behaviour in the TLB system.

To retrieve the physical address of a virtual address passed to the helper, the *get_physical_address* function is called. This function is part of the pre-existing MIPS TLB, and is supposed to lookup a virtual address in the TLB and provide the physical address, or return with an error indicating why the address could not be retrieved. However, for some virtual addresses, it returns errors of either TLBRET_NOMATCH or TLBRET_INVALID. This is possibly due to the virtual address not actually being in the TLB when this function is called, and thus there is no corresponding physical address. However, in the time constraints given, it was not possible to determine how to delay calling this function until after the TLB was properly loaded.

Currently a work around is in place where if the TLB is unable to find a physical address the access is simply ignored, as this only happens to a small percentage of accesses, and thus shouldn't affect results greatly. However, this remains something that should be investigated and fixed properly.

Chapter 8

Software Testing

It is well known that testing software thoroughly is essential for success, and, although time constaints were tight, the system was tested in various different ways.

8.1 Overview of Techniques Used

In many cases, it was fairly clear from the data being produced when the cache was working or not. However, for more intricate parts of the system, such as cache instructions, it was hard to know whether the cache was behaving correctly by simply analysing the logfiles.

As the cache system was designed in a modular way, it was possible to build a skeleton around it such that the algorithms could be tested without the constraints of QEMU. This testing platform is stored in the *testbench* folder inside *target-mips*, and is essentially comprised of a Makefile and a main 'runTests' file which is used to set up the testing environment. Within this file, is is fairly simple to initialise different caches in whatever configuration is desired, and then directly call cache manipulation routines on them. It made it possible to specifically test different parts of the algorithm, and made debugging significantly easier.

8.2 Testbench

The testbench described was implemented by essentially constructing a virtual QEMU environment in which the system could run. This was possible because none of the modules directly depended on QEMU, except in one case. Several of the methods in *mips-cache-opts.c* made use of the *pstrcat* and *pstrcpy* functions that QEMU provided, as the official QEMU 'Hacking' guide recommends their use instead of *strncat* and *strncpy*

[4]. These two functions are essentially more robust versions of their standard library counterparts, which is why they are preferred in QEMU. However, in order to compile the testbench separately from QEMU, this dependency had to be severed. This is accomplished by only including the QEMU headers in *mips-cache-opts.h* when a preprocessor symbol (*GDP17_TESTBENCH*) is not defined. When this symbol is defined (by the Makefile in *testbench*), prototypes for these functions are included instead. They are then implemented in *runTests.c*, as simple wrappers around the standard library versions.

8.2.1 Building the Testbench

The testbench Makefile builds the testing environment, and performs several tasks to do this:

- Defines *GDP17_TESTBENCH*
- Includes and links with glib-2.0
- Creates object files for all cache system modules
- Creates object files for each test file
- Links all modules together to produce *runTests*

As several parts of the system use GLibC functions (such as *g_malloc0*), the glib-2.0 libraries are required. However, they are also required to build QEMU so this should not be a problem. In addition, *pkg-config* is used to automatically configure the include and link paths. Therefore, building the testbench is done by simply typing *make* while in the directory.

8.2.2 Using the Testbench

The main file in the testbench, *runTests*, is used to declare the caches required, run a series of tests on the caches, and then dump the cache data for analysis. Adding tests is done in two ways – either by adding the tests directly to *runTests.c*, or by creating a new file to hold the tests. If the second method is chosen, the functions created must be declared in *runTests.h* so that they are accessible from *runTests.c*, and added to the TEST_OBJS variable in the Makefile so that they are compiled.

Chapter 9

Cache Performance Analysis

This chapter looks at how cache performance can be measured, presents an overview of how the Python programming language is used to analyse cache data, and finally presents some real results of applying these methods. This includes a set of graphs that show various performance statistics of different cache designs and different applications.

9.1 Terms associated with Cache Performance

There are many factors that are used to judge the performance of caches. Some of the terms used to analyse cache performance are explained below [13]:

Hit time: The time taken to access a particular piece of memory in the memory hierarchy and produce a hit.

Hit rate: It is an indication of how many memory accesses are found in a particular level of a memory hierarchy. It is the ratio of the number of hits to the total number of cache accesses. This is a good indication of how effective the cache is as it gives an indication if the cache is being effectively utilized. Equation 9.1 shows how the hit rate can be calculated [13].

$$\text{Hit Rate} = \frac{\text{Hits}}{\text{Hits+Misses}} \tag{9.1}$$

Conversely, the miss rate indicates how often the cache misses and is shown in 9.2:

$$MissRate = 1 - HitRate \tag{9.2}$$

Miss penalty: It is the time required to fetch a cache line from the lower level in the memory hierarchy to the requested level. This also includes the time needed to read the line once it is retrieved from the lower level of the memory hierarchy.

9.2 Factors affecting cache performance

Misses in a cache can be classified into three categories which are discussed below [11, 13, 17]:

- **Compulsory Miss:** This miss occurs the first time the cache is accessed. Since the requested address never existed in the cache, a miss occurs.
- **Capacity Miss:** A capacity miss occurs because the cache was full. A bigger cache can be used as a solution but there is trade-off with increasing the cache size as hit time increases.
- **Conflict Miss:** Here, two or more addresses compete for the same cache line or set due to a similar address mapping. A cache line gets replaced by another even if the cache still has space. Usually, associativity is used to decrease collision misses as data can be placed in more locations and a fully associative cache completely eliminates conflict misses.

It can be observed that the main factors affecting performance are the hit rate and latency associated with hits or misses. This project looks at improving cache performance by analyzing the hit rate by looking at its size, associativity, block size in a line and the replacement strategy used for associative caches.

9.3 Ways to improve cache performance

The hit rate of a cache can be improved by suitably adjusting the parameters mentioned earlier. There is a trade-off and a limit to which anything can be adjusted.

Cache size should be large enough to hold the instructions that are needed for execution by the processor to decrease the miss rate. Yet, at the same time a large cache has a higher hit time due to an increased access time and also takes up more space. This is why L1 caches are smaller than L2 caches as they are usually next to the processor on chip and do not occupy a large area [11, 13].

Block size of a cache affects the miss rate. The miss rate decreases as block sizes are increased to a certain extent. After a certain point the miss rate increases as spatial locality decreases with a large block size and also the miss penalty increases as it takes more time to fetch a large block of data on a miss [13].

Associativity makes more efficient use of the space available in the cache. The main benefit of associativity is that data can be mapped onto more than one slot which increases the hit rate of the cache. Therefore when two addresses map to the same set there is no collision and this makes more efficient use of the space available in the cache [11, 13]. But if associativity is used the cache also needs to implement a replacement algorithm to determine which blocks in the cache to replace on a miss. Also, more than one data block has to be compared for a hit and therefore due to these factors there is additional overhead on the cache [11, 13].

The replacement strategy used in an associative cache can also affect the hit rate. A more complex strategy could improve hit rate but would require extra logic in the cache and may at most times be unnecessary [11, 13]. A simple strategy, such as Random Replacement, may sometimes replace a line that is used quite often and therefore this degrades the cache performance [11, 13]. So depending on the application the replacement strategy should be carefully chosen to get optimum results.

Prefetching data is another way to improve the hit rate of the cache. Data is fetched before the execution of an instruction stream based on some hint from the processor. Since data is fetched before it is needed, prefetching can reduce the number of compulsory misses in the cache and thereby improve the performance [11, 13].

Another way to improve performance is by using buffers. A write buffer could be used to queue writes to main memory which usually takes longer and this reduces latency [13].

9.4 Analysis and Plotting with Python

For each cache design tested, a CSV formatted log file is created, which contains the hit and miss counts for each line in the cache. This allows the hit/miss ratio (the hit rate) to be calculated for the entire cache, to be compared with the hit rate of all the other designs tested.

9.4.1 Performance Analysis

To analyse each cache design, the QEMU MIPS executable should be started with the appropriate cache configuration, and an application started. In order to judge cache performance, a set of common programs were used as 'benchmark' applications. These were:

- bzip2, a data compression tool.
- gnuchess, a chess game engine.
- imagemagick, a set of image manipulation tools and libraries.
- stress, a tool which imposes a high load on the cache.
- standard Linux tools (grep, sha, find, diff)

Running these programs in QEMU system mode is done by modifiying the Linux startup scripts, as explained in Section 4.3.1. Once the system is set up to start the correct program, a script called *automated_test.sh* is used to repeatedly boot the system and let the program run to termination, but with different cache configurations on each boot. This results in the working directory filling up with log files, which need to be sorted before anything useful can be done with them. As explained previously, the name of each file corresponds to the cache configuration used, in order to facilitate the sorting process.

Figure 9.1 shows an example log file of an I-Cache. The fields are the cache *line number* and the corresponding number of *hits* and *misses* per line. However in the case of a data cache, it is necessary to differentiate between Store hits or misses, and Load hits or misses. So in this case the parameters are: cache *line number*, store *hit*-count, store *miss*-count, load *hit*-count and load *miss*-count as shown in Figure 9.1(b).

	A	B	C	D
1	**Cache Line**	**NO, Hit**	**NO, Miss**	
2	0	2090989	3610052	
3	1	2099714	3614956	
4	2	3253452	3580524	
5	3	3247945	3577877	
6	4	3558363	3561085	
7	5	2377999	3566367	
8	6	2379872	3560675	
9	7	2402853	3465311	
10	8	1679428	2987721	
11	9	1768993	3021301	
12	10	1767441	2995442	
13	11	1605447	2850115	
14	12	1605135	2856996	
15	13	1574740	2826394	

(a) I,L2 Cache CSV

	A	B	C	D	E
1	**Cach Line**	**Store NO Hit**	**Store NO Miss**	**Load NO Hit**	**Load NO Miss**
2	0	121578	57442	287837	202936
3	1	89273	44454	627662	80274
4	2	77702	40594	152035	138815
5	3	75055	53587	106731	95169
6	4	82601	45170	667485	185026
7	5	147847	42592	219085	128221
8	6	114895	44156	110092	58143
9	7	62366	38885	95703	50210
10	8	65543	39211	105560	61031
11	9	69772	36085	191068	59515
12	10	63191	36474	120203	60307
13	11	68308	37674	79755	56930
14					

(b) D Cache CSV

Figure 9.1: CSV File view

The Hit Rate of the cache can be derived using these measured parameters. The sum of *hit*-count and *miss*-count indicates the total number of accesses which has been made to each cache line. Therefore, for each cache line the Hit Rate is found by applying

Equation 9.3. Furthermore, the overall Hit Rate of each cache design can be found using Equation 9.4.

$$LineHitRate = \frac{Hit - count}{Hit - count + Miss - count} \tag{9.3}$$

$$OverallHitRate = \frac{Sum(Hit - counts)}{Sum(Hit - counts) + Sum(Miss - counts)} \tag{9.4}$$

9.4.2 Log File Sorting

Each test program should be simulated for all possible cache designs. Considering the number of cache parameters that can be changed, this would produce 420 different CSV files. All CSV files produced should be classified according to their name, and moved to the same place for analysis. For this purpose a Python script was written, which will now be explained.

The primary module which performs the file organisation and analysis is called *QEMU-off-line-Analysis.py*, and is intended to be executed from the command line. It executes the *Log_File_Sorter.py* Python script which performs the main log file sorting work. The script first prompts the user to input the path to the folder for which the log files are located – this will be the same folder that QEMU was run from. Then, in the same folder, it creates 'L1' and 'L2' folders, and a number of subfolders named for the different parameters that may be analysed.

- **All: This folder is used to compare all cache designs of a certain kind.**
- **Associativity: This is used to compare cache associativity while cache size changes.**
- **Block-Size: Compares cache block-size while cache size changes.**
- **Replacement: Compares the cache replacement strategies while cache size changes.**

Figure 9.2 shows an overview of how folders and files are arranged in each test folder using *Log_File_Sorter.py*.

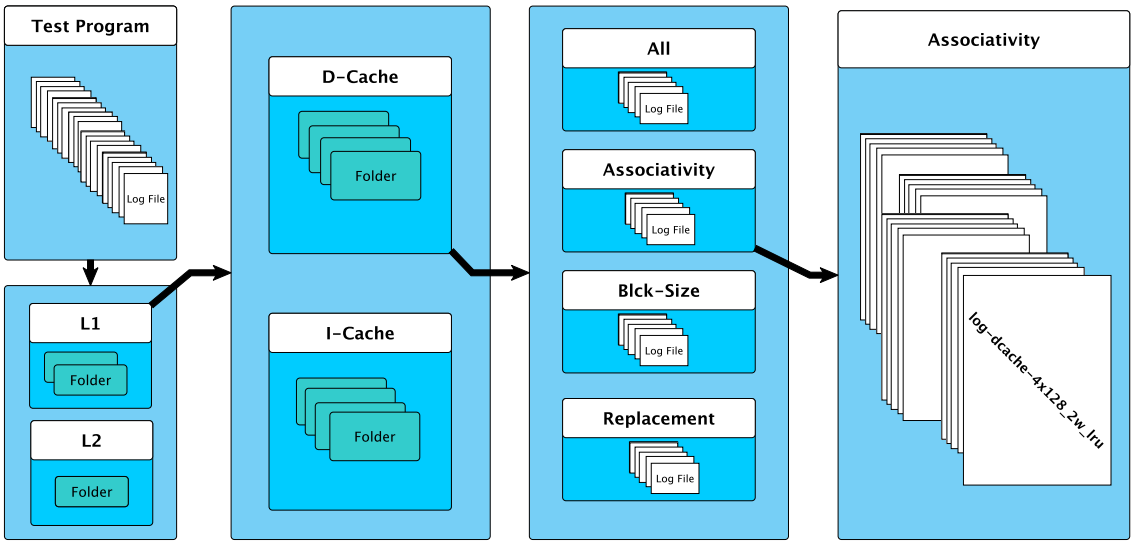


Figure 9.2: File and folder arrangement after sorting algorithm is applied

Once the log files are sorted, *QEMU-off-line-Analysis.py* uses certain functions to get the name, path and data within each log file. Then it uses other methods and modules to analyse the data and creates three different graphs.

The names of all the CSV files within the given folder are found by the following methods: *Get_File_Names(Folder_P)*, which gets the name of all the files in the given folder, and *Find_csv_filenames*, which in turn filters only the file names with *.csv* extensions. An error message is displayed if an exception occurs here.

Next, the log file name and content are analysed. For this purpose a dictionary is defined (named *DicFiles*), in order to make it easer to access and organise the data. This dictionary is essentially a mapping between log files, corresponding to certain cache designs, and the statistics extracted from the log file. The dictionary 'keys' contain the exact file name, and the 'values' contain the following information.

- Cache Size
- Cache Associativity
- Cache Replacement strategy
- Cache Block-size
- Cache Hit rate
- Cache Miss rate
- Cache Store Hit rate (only D-cache)
- Cache Store Miss rate (only D-cache)
- Cache Load Hit rate (only D-cache)
- Cache Load Miss rate (only D-cache)

The dictionary 'keys' and 'values' are found using two methods. One of these is *Find_Cache_Properties(File_name)*, which populates the first values of the dictionary.

This method takes a cache file name as the input argument. The cache size, associativity, block-size and replacement strategy of the cache design are then found through string manipulations done with file name. This information gets passed to the calling function. The second method is called *Cache_analyse(Path_FileName)*, and takes a file path and a name as its arguments. It opens the given file and goes through each row in the CSV file, extracting the relevant data (shown in Figure 9.1(a) and 9.1(b)) from each column into different lists. The mathematical equations explained in Equation 9.1 and 9.2 are then applied to these lists to find the hit and miss ratios. These results are returned to the calling function and inserted into the dictionary.

9.4.3 Comparison Algorithm

The result of the last section is a dictionary which contains all necessary data in it's keys and values to perform analysis of the data. The next step is to find out which parameters of the cache are being analysed. For simplicity, this is pre-defined by the name of the folder which the CSV files are placed in (shown Figure 9.2). For example, if they are within the associativity folder, then the associativity of different cache designs is being compared.

For plotting and analysis purposes, it is necessary to choose which parameters will be compared, since it pointless (and very hard) to compare all cache parameters on one graph. It was decided that one of the important properties that should always be considered is the cache size. The other three parameters that can change between cache designs are the Associativity, Block-Size and Replacement strategy. Thus, on a 3D graph, it is possible to display the hit rate on one axis, the cache size on another, and one of these parameters on a third. Figure 9.2 shows these folders and also an extra folder named 'All' which is created for each cache level and type along with the rest.

The folder All contains all designs of a certain cache type in each level. For example, *stress-test/D-cache/All* would hold the log files of all data cache designs that are used in stress-test simulations. The log files in this folder can be compared to find the top ten best performing cache designs in each level and category.

9.4.4 Plotting

In order to make the final results and cache design comparisons user friendly, the plotting is done in three different forms. This also helped in testing and debugging the python code. **3D**, **bar** and **line** graphs are plotted using the Matplotlib Library. Each graph plots the same information but represents it in a different way, in the hope that interpreting the data is made easier.

In order to make the plotted figure easy to understand, the parameters on each axis should be in a certain order (ascending / descending, for example). An easy way to overcome this problem was to create a list for the **x** and **y** axes, which can then be sorted algorithmically. The **z** axis always shows the hit rate.

Listing 9.1 shows how these lists may be assigned at the start of the main module.

```
ListL1sizes = [2,4,8,16,32]
ListL2sizes = [256,512,1024,2048,4096]
ListBlockSizeL1=[1,2,4,8]
ListBlockSizeL2=[4,8,16,32]
ListAssociativity = ["DM","2-Way","4-Way"]
ListRep = ["None","LFU","LRU","Random"]
```

Listing 9.1: Assigning list values

In the dictionary, the Hit ratios (values) of log files (keys) have to be allocated to the graph axis variables before they can be plotted. For this purpose the values and two axis parameters are sent to a method which arranges this data in the correct order in a 2D matrix and returns the matrix back to the calling function. This method is named *Arrange_3DMatrix(FixedList1,FixedList2,zList)*. The arguments FixedList1 and FixedList2 contain the X and Y axis parameters. For instance FixedList1 could contain the associativity and FixedList2 could hold the L1 caches sizes. zList is a List that contains the data required to plot the graph – values for the **x**, **y** and **z** axes. These values hold information such as Hit-rate, Miss-rate, Load-Hit-Rate, Load-Miss-rate etc, and are used for plotting the graphs according to the given Fixed Lists in future steps of the program. Figure 9.3 shows how the values in *FixedList1*, *FixedList2*, and *zList*, map to a possible graph.Listing 9.2 shows an overview how the Python code executes this algorithm.

```
def Arrange_3DMatrix(FixedList1,FixedList2,zList):
    DMatrix =[[0]*(len(FixedList1)) for i in xrange (len(FixedList2))]
    length=0
    for i in range(len(zList)):
        DMatrix[FixedList2.index(zList[i][1])][ FixedList1.index(zList[i
    ][0])]=zList[i][2:]
    while length==0:
        for mem in DMatrix:
            for mem2 in mem:
                if isinstance(mem2,list):
                    length=len(mem2)
    print length
    for mem in DMatrix:
            for mem2 in mem:
                if not isinstance(mem2,list):
                    mem[mem.index(mem2)]= [0]*length
    return DMatrix
```

Listing 9.2: *Arrange_3DMatrix(FixedList1,FixedList2,zList)*

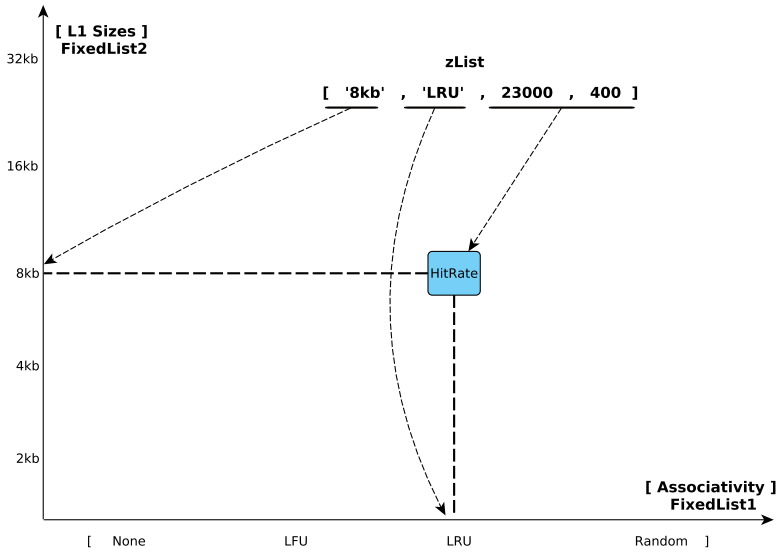


Figure 9.3: Figure showing how different arguments passed to *Arrange_3DMatrix* are used

3D Plotting is implemented in another module, called *CachePloting3D.py*. The parameters passed to this module for plotting includes the two lists specifying the **x** and **y** axis, and the third parameter is the 2D matrix mentioned earlier, which holds the **z** axis values. However, in the first 3D graphs that were plotted the differences between Hit Rates were not apparent due to the small changes between designs. This is illustrated in Figure 9.4.

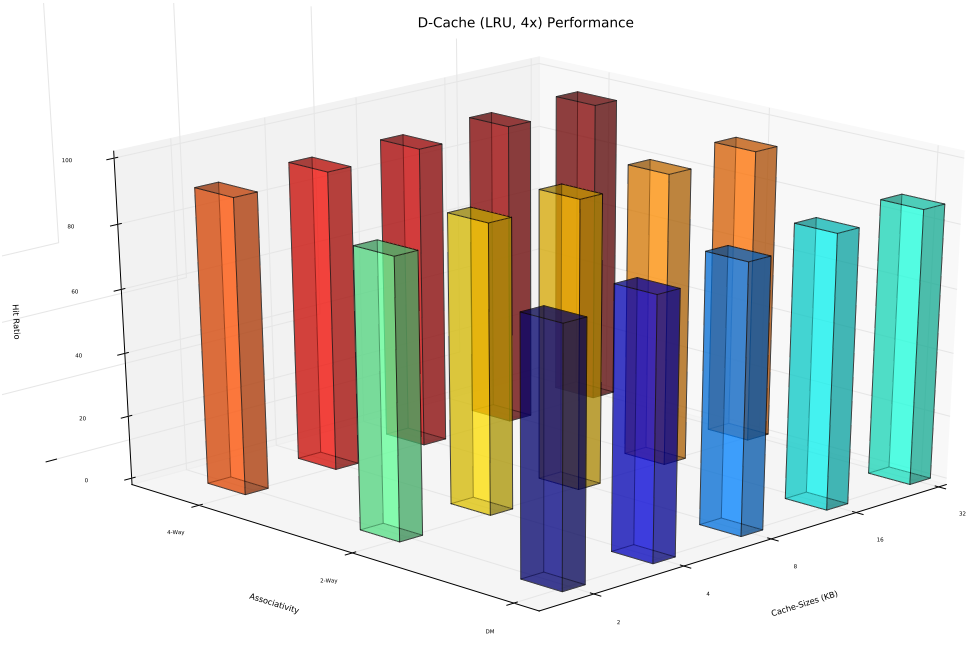


Figure 9.4: Normal 3D view of D-Cache

In order to make analysis easier, the minimum value of the **z** axis was adjusted to the lowest performing design, i.e., the plot is normalised. The difference between each performance value and this minimum value is then multiplied by 100 to magnify the difference in the 3D plot. Figure 9.5 shows an illustration of the same analysis as before, but with this algorithm applied as well.

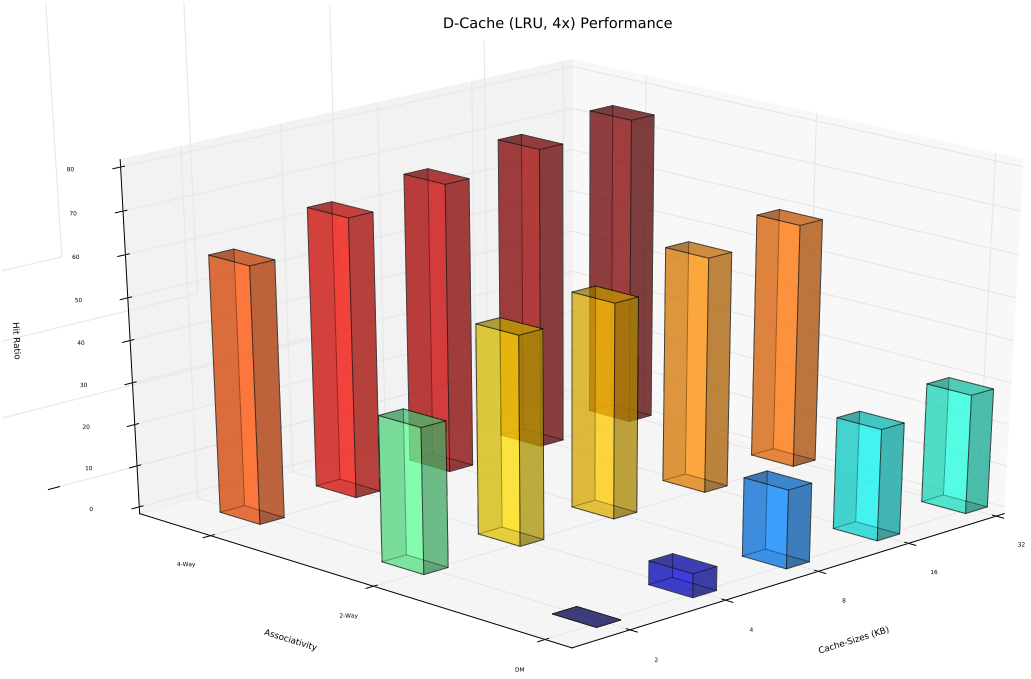


Figure 9.5: Scaled 3D view of D-Cache

Note: The difficulty which was faced at this stage was that using this algorithm resulted in having one value of the **z** matrix set to zero and this caused an error when saving the resulting figure as a vector graphics type file. After debugging the code, it was concluded that Matplotlib, when 3D plotting, has a problem with displaying the value on the plot if *dz* is zero. The zero values resulted in 'NaN' of 'infinite value' points in the plotting process. To solve this bug a small value of 10^{-12} was added to each value of the **z** matrix. This small value has an insignificant effect on the final results shown, but it allows the graph to be saved in any vector graphics format.

In the case of plotting **Bar Graphs**, the same values for **x**, **y** and the matrix holding **z** are passed to an instance of the python script called *CachePloting2DBar.py*. However, only two axes are used so the parameter which is being monitored is shown on **x** axis and the **y** axis shows the Hit and Miss ratios. Nevertheless, to make the graph more informative, the other parameters and also numerical values of Hit ratio are displayed on the bars. Figure 9.6 shows an example of a Bar Graph for the best D-cache performances in the stress test.

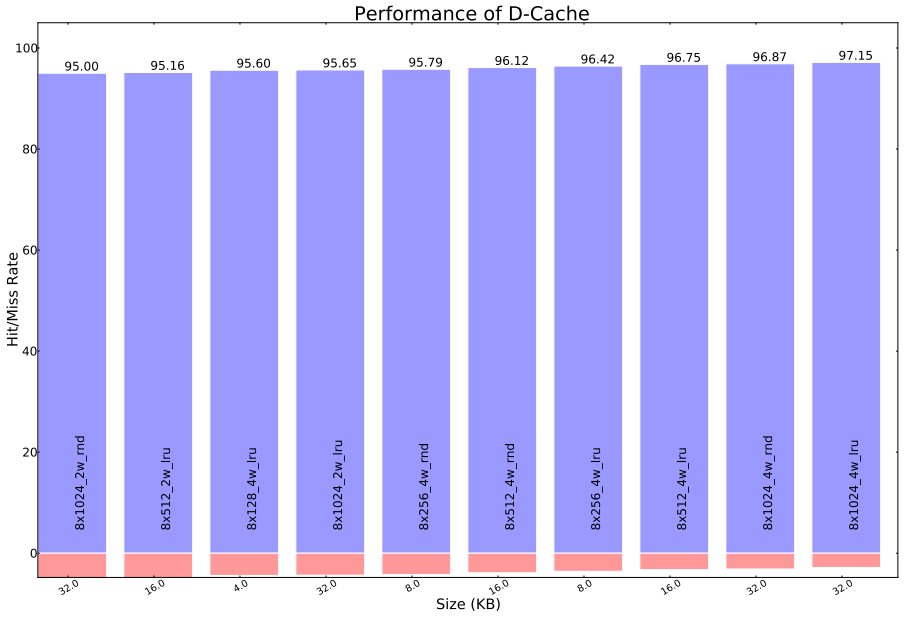


Figure 9.6: Bar Graph view of D-Cache

Line Graphs are generated by passing the same lists and the **z** matrix to an instance of the *CachePloting2DLine.py* module. The values held by zMatrix are placed in a list of lists, where the sub-lists hold the Hit Ratio of each cache size. Then, using a loop operator over values in these lists, a plot of cache Hit Ratio against a particular cache parameter is plotted (i.e., different associativity or block-size). However, each list is checked prior to plotting so that if any hit-rate value is zero, it is discarded. It was found during debugging that if some cache sizes are missing from the log file this code would generate zeros on the plot for these sizes. Figure 9.7 show an example of plotting Hit Ratio against replacement strategy for three log files.

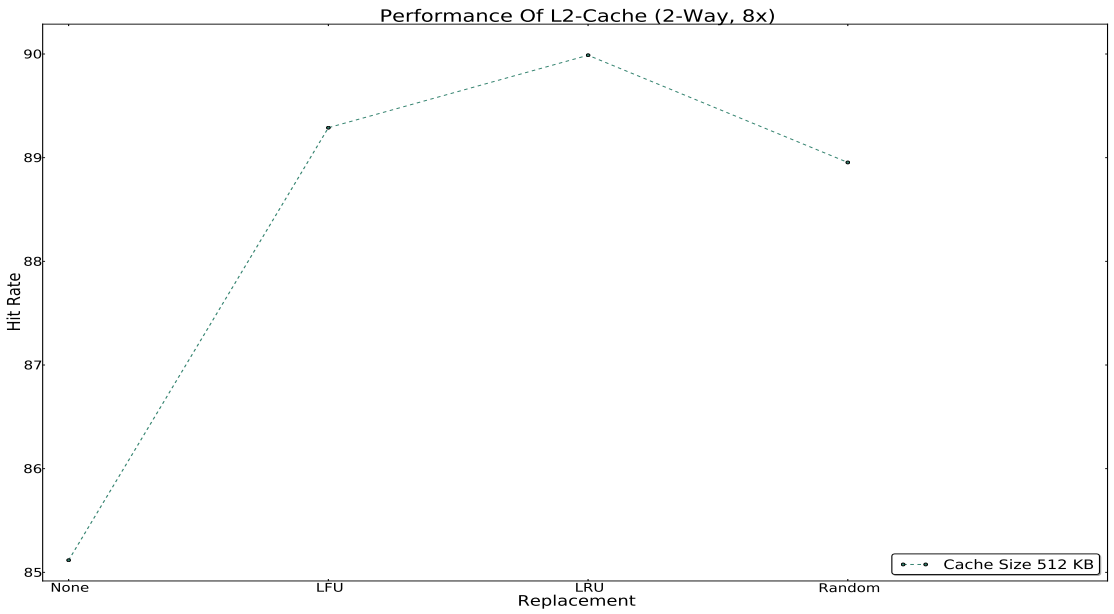


Figure 9.7: Line plot view of L2-Cache for four log files only

9.4.5 Debugging and Testing

Debugging the Python code was done in two main ways. Firstly, the 3 different plots (3D, Bar, Line) of a set of data were compared, which proved to be a useful tool to analyse the integrity of the data as it passed to different methods in the python code. Additionally, exceptional cases such as missing files, data-type errors, and similar bugs were debugged using breakpoints and a mature Python IDE (Spyder[1]).

Secondly, an Excel spreadsheet named *Cache_Profiling_Performance_Analysis.xlsx* was created, which performs essentially the same tasks that the python code does, except that the data must be entered manually. In the 'Performance' sheet, the result of analysing the log file data is displayed. The graphs produced were then compared with the ones created by Python, and checked for inconsistencies that would indicate a bug in the code. Figure 9.8 and Figure 9.9 show a graph produced by the python code, and by the excel code respectively, for the same log file.

[1]https://code.google.com/p/spyderlib/

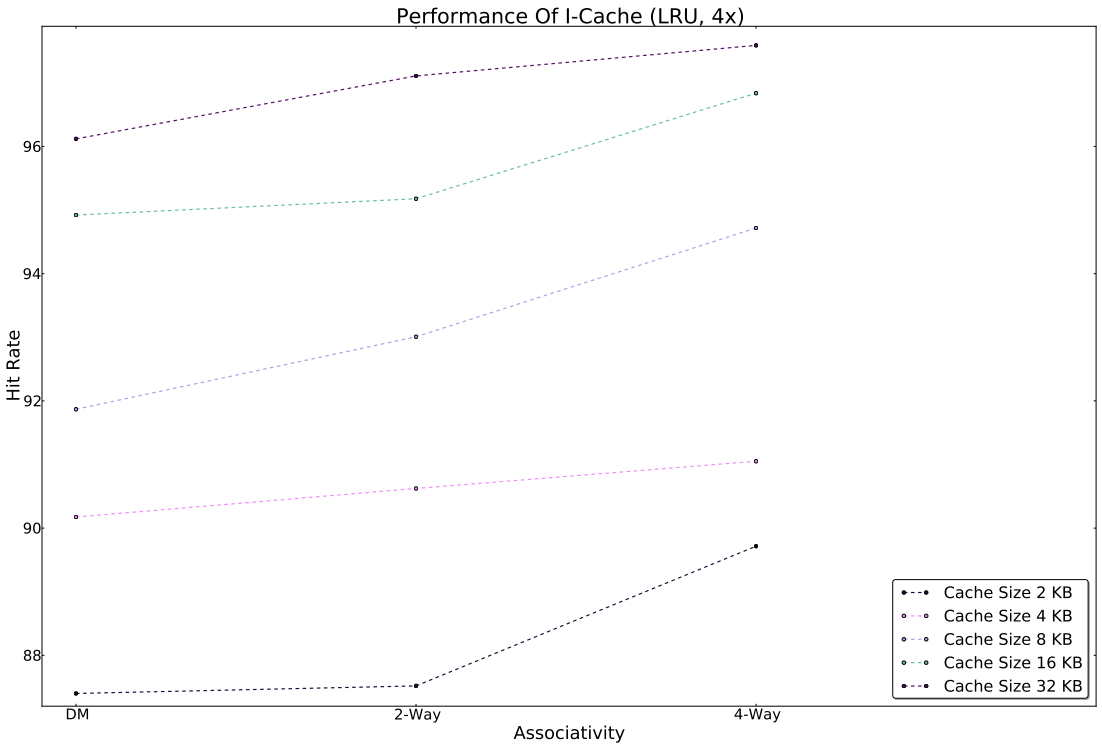


Figure 9.8: Graph created by the Python code

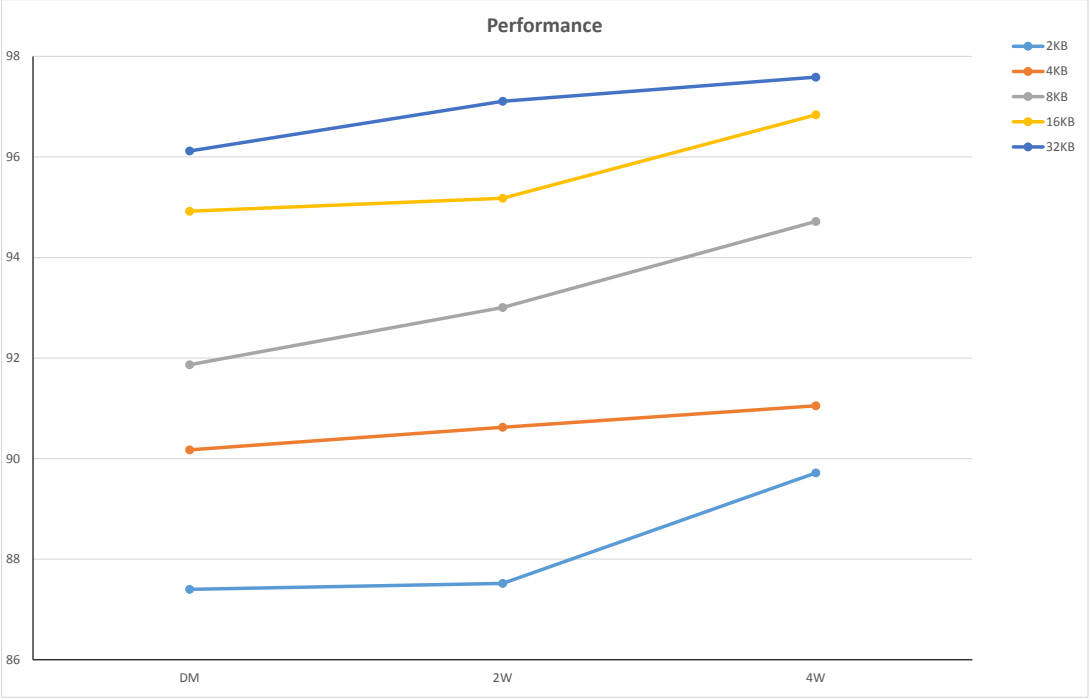


Figure 9.9: Graph of the same data, created by the Excel code

9.5 Cache Performance Analysis

This section shows how different cache configurations performed against different test programs. In each case the hit rate was plotted against two varying cache parameters. The size of the cache was varied in all cases. The other parameter that was varied was block size, associativity or the replacement strategy used in the case of an associative cache. When two parameters were varied the other parameters were kept constant and the values used have been mentioned in the relevant sections. The best possible design considerations are also shown for any particular test program used. The constant values are listed below:

- **Block Size:** 4 words for L1 cache and 8 words for L2 cache.
- **Associativity:** 2-Way set associative cache.
- **Replacement Strategy:** Least Recently Used (LRU).

The test programs used were Stress (test ran for 10 seconds), SHA sum on a 10MB file (used to verify integrity of files) and a sort on a 56kB file. These programs effectively tested if the caches were properly implemented and gave a gauge into the hit rates when the caches were tested with a mix of light and intensive programs. The caches used a write through policy on hit and a write allocate policy on miss. There was no latency between levels of memory hierarchy taken into consideration in this software simulation.

Size of the cache changed for all the cases along with one of the above mentioned parameters. The data cache and L2 cache also show the number of load and store hits for different configurations which are annotated on the respective graphs. The following sections discuss the different results produced by varying the cache parameters mentioned.

9.5.1 Associativity

The cache associativities were varied to plot the graphs in Figures 9.10 - 9.18. The cache was either direct mapped, 2-way set associative or 4-way set associative. It can be seen that 4-way set associative cache has the best performance in all three cases and this was expected as data has more flexibility to be placed in the cache as it can be held in four possible locations. Therefore, the hit rate of the cache is higher as compared to a 2-way cache where data can be placed in two possible places and a direct mapped cache where data can be placed in one location.

Bigger cache sizes usually gave better performance in all cases. For the L2 cache, a size of 1MB upwards gave a better performance for a direct mapped implementation as compared to the other sizes tested. Bigger cache sizes averted capacity misses and therefore improved the hit rate. Although, it can be seen that some sizes that were close to each other gave almost comparable performances. For example, in the sort program in the L2-Cache, a 4-way set associative 2MB cache was almost as good as a 4-way set associative 4MB cache and so a bigger size in this case wouldnt be beneficial if we had to choose between the two sizes.

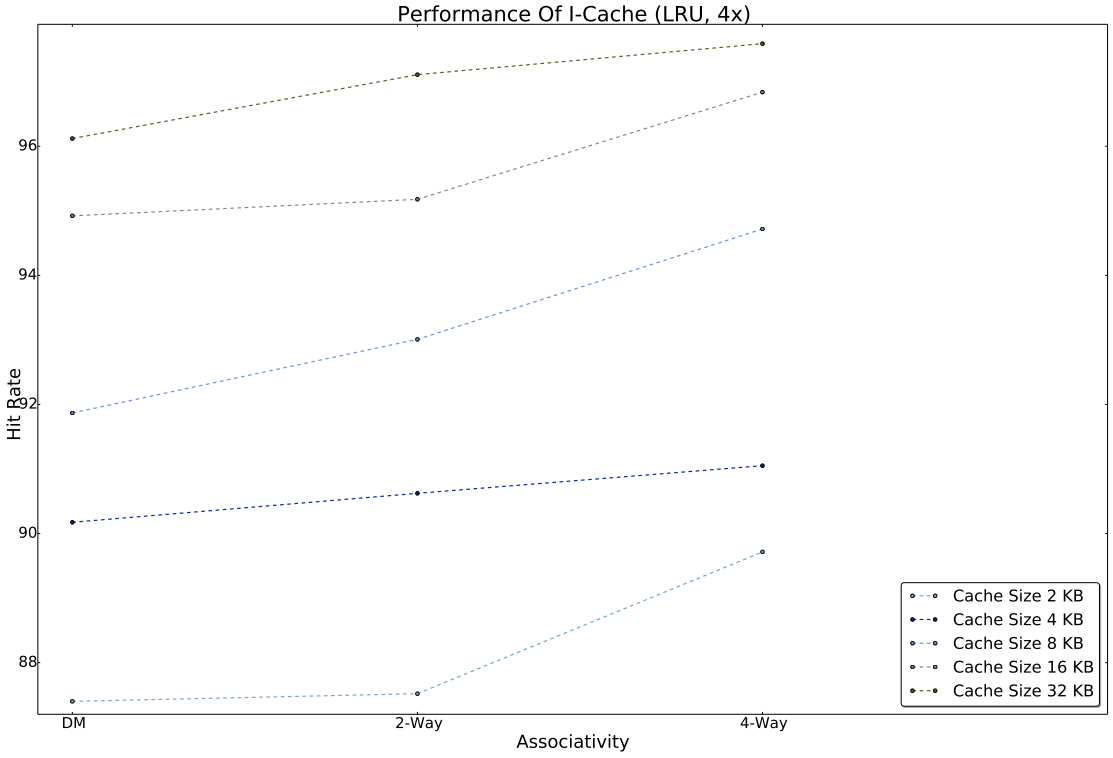


Figure 9.10: Stress test on I-Cache configurations.

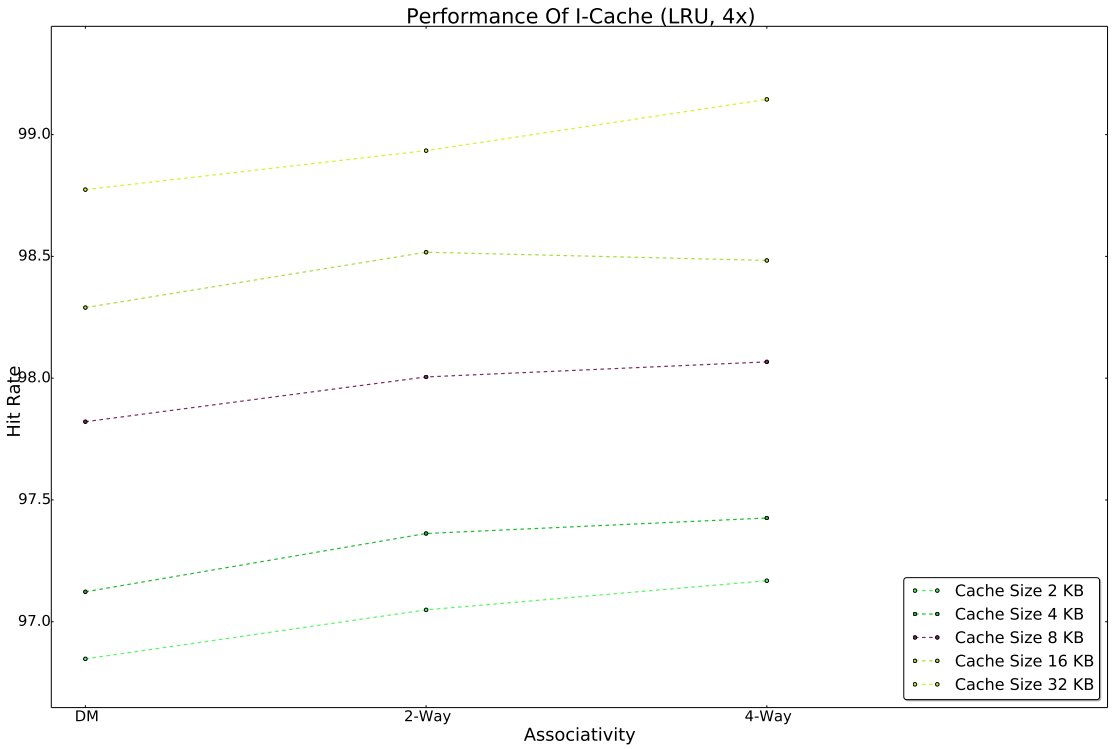


Figure 9.11: SHA test on I-Cache configurations.

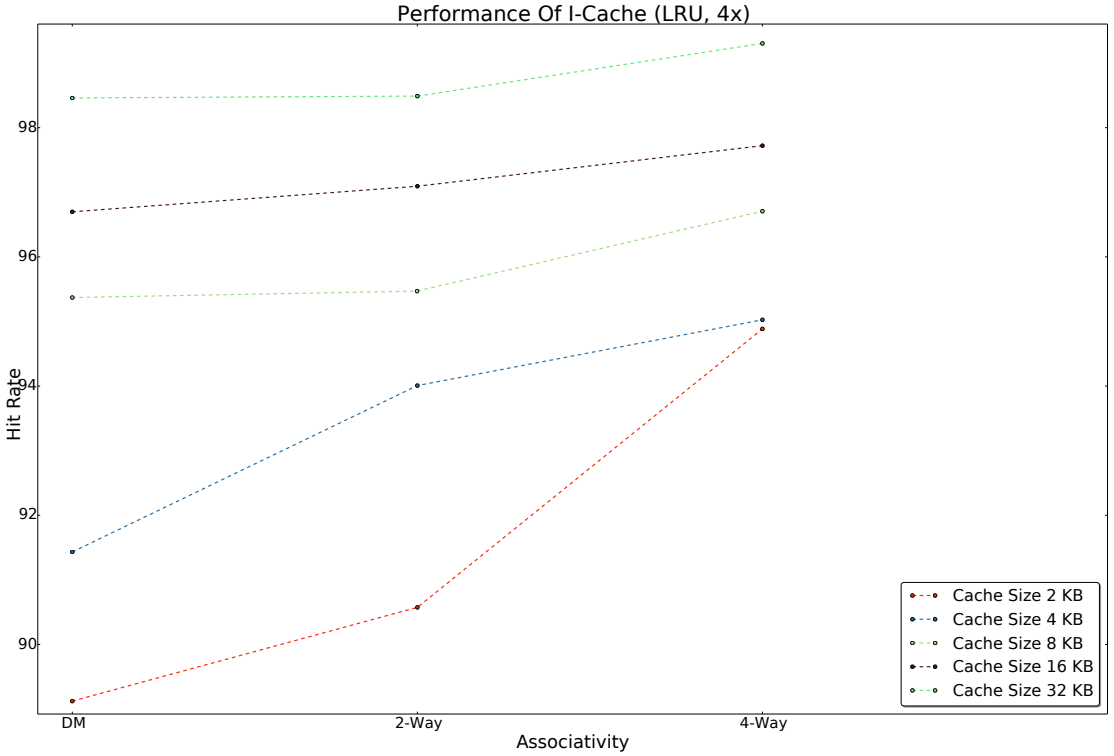


Figure 9.12: 56kB sort on I-Cache configurations.

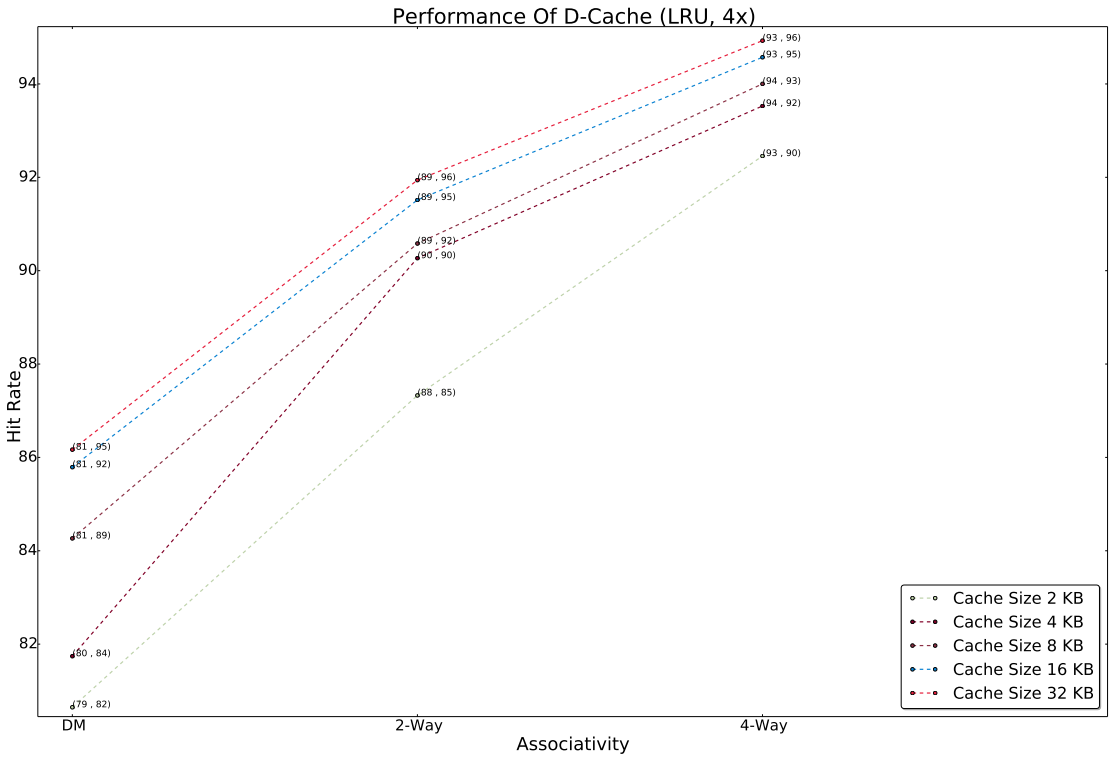


Figure 9.13: Stress test on D-Cache configurations.

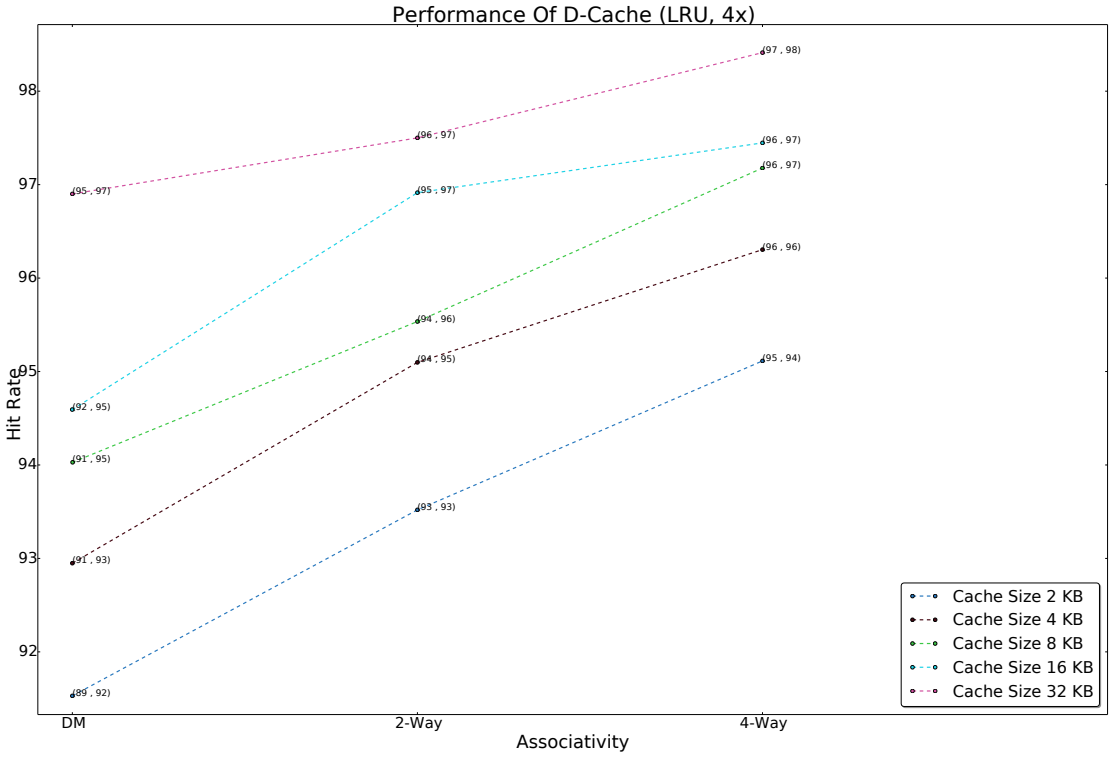


Figure 9.14: SHA test on D-Cache configurations.

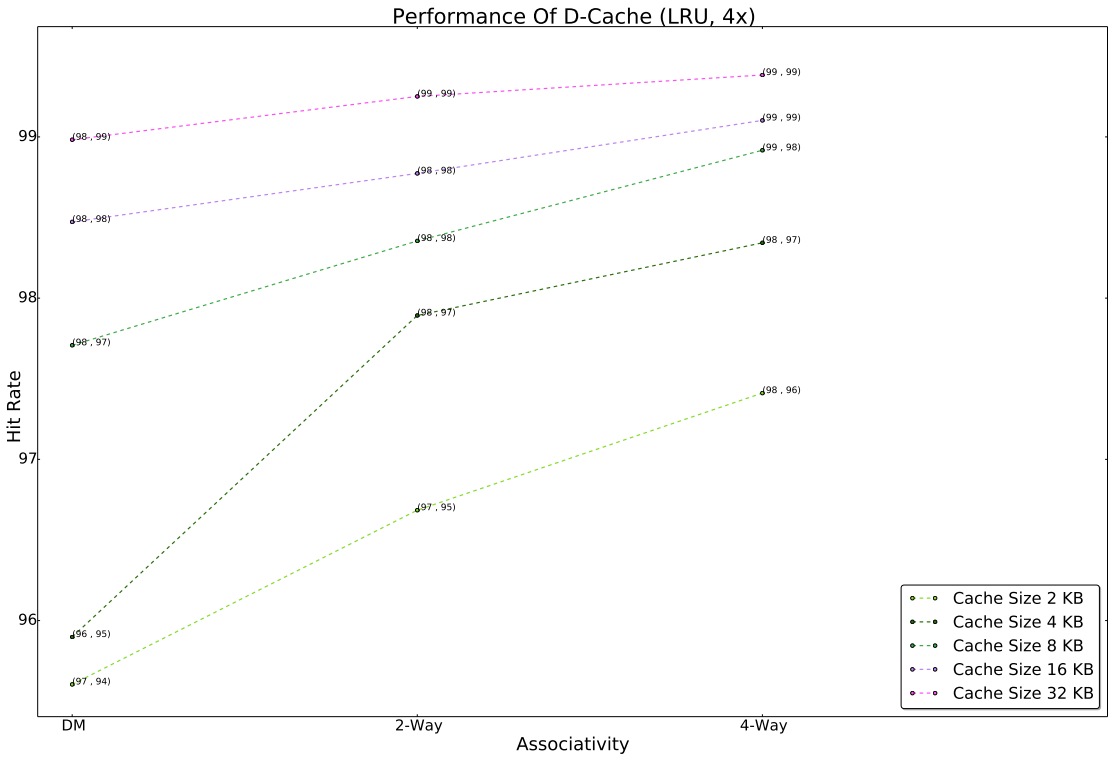


Figure 9.15: 56kB sort on D-Cache configurations.

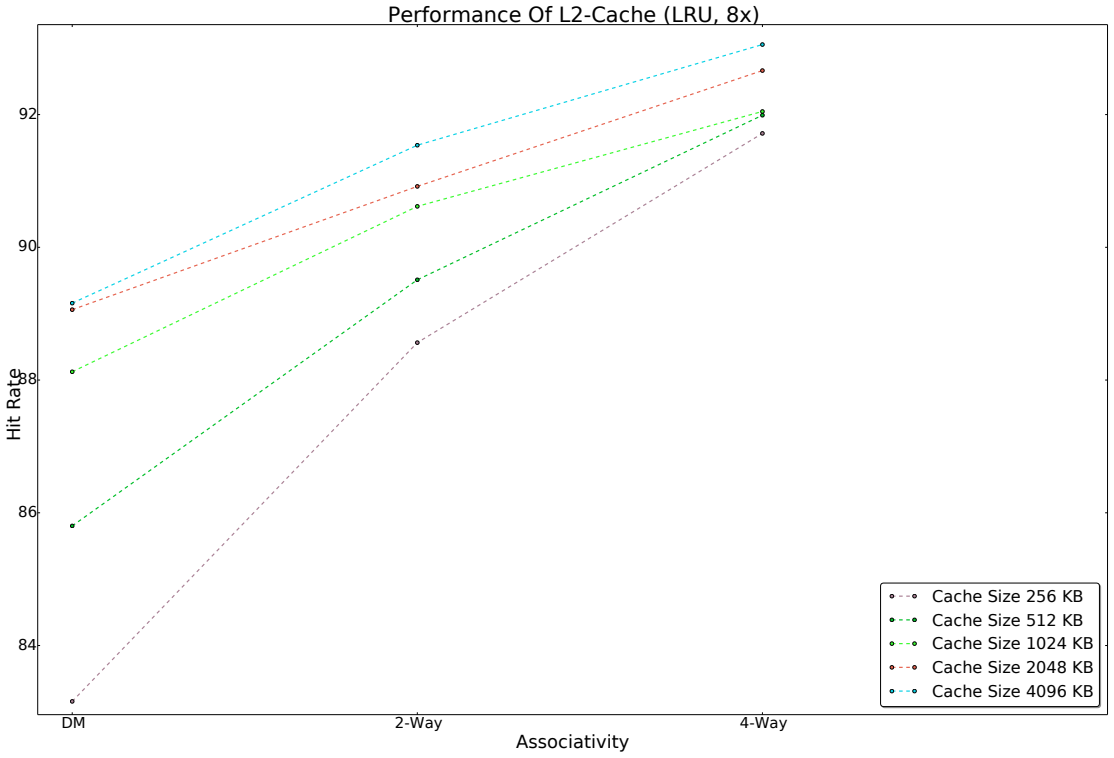


Figure 9.16: Stress test on L2 Cache configurations.

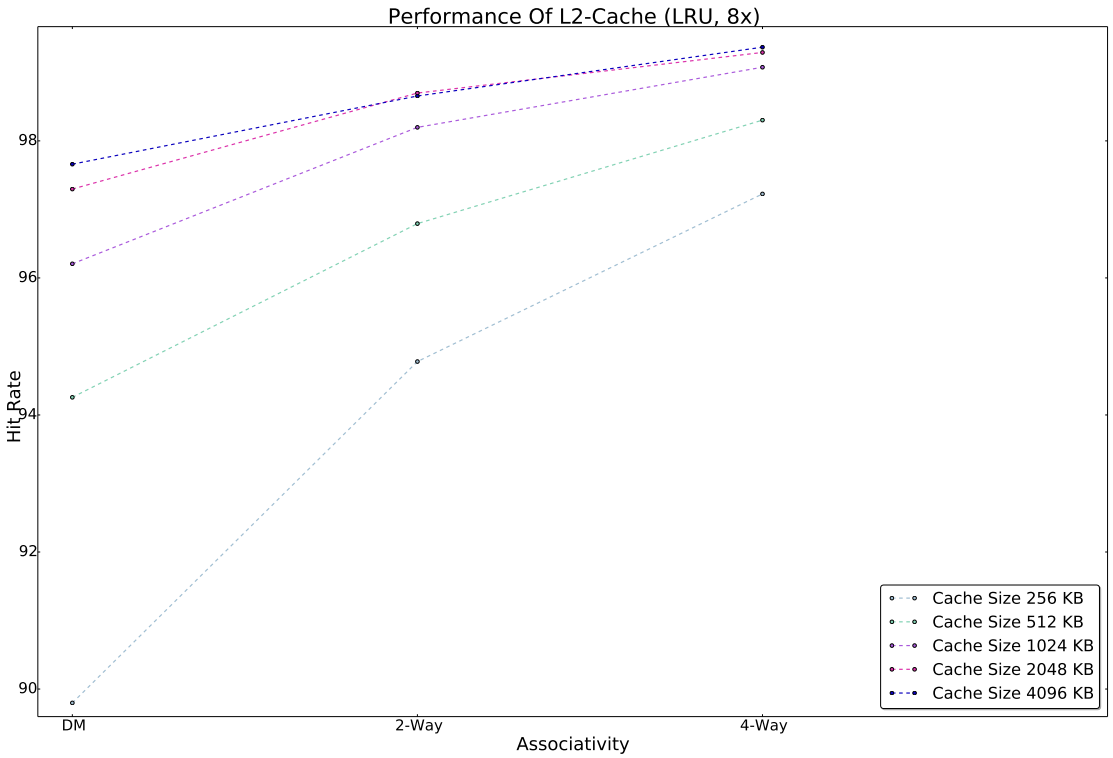


Figure 9.17: SHA test on L2 Cache configurations.

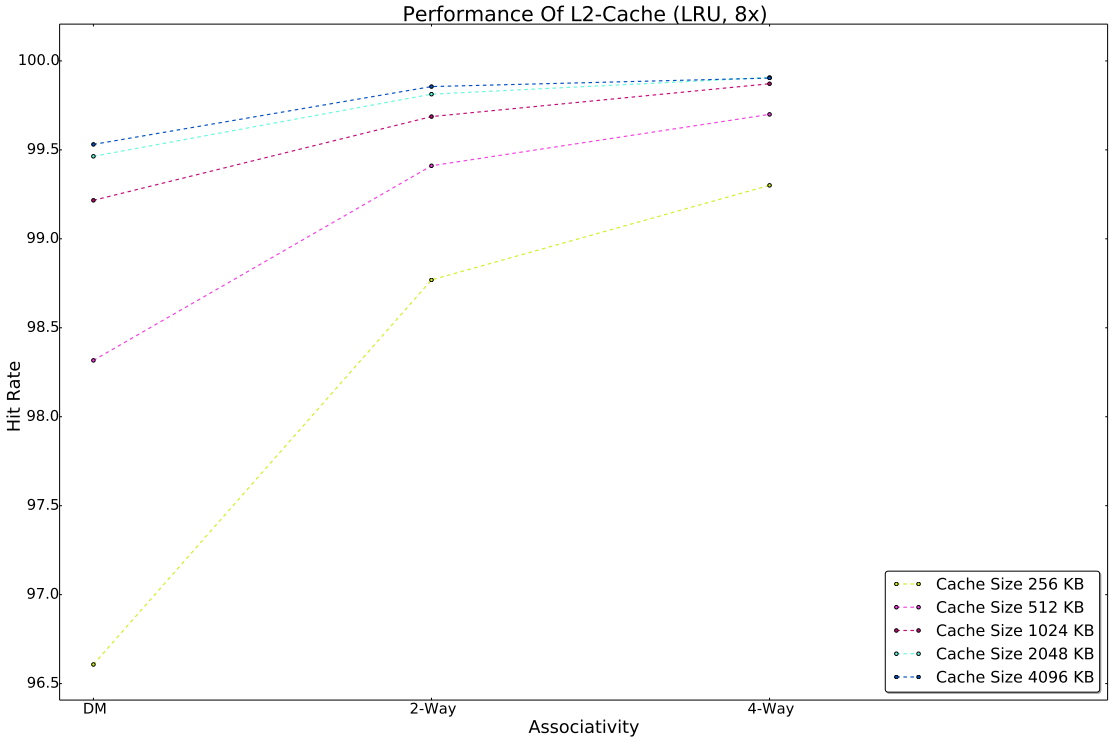


Figure 9.18: 56kB sort on L2 Cache configurations.

9.5.2 Replacement Algorithm

The results of different replacement algorithms are shown in Figures 9.19 - 9.27. It can be observed that Least Frequently Used (LFU) performed the worst amongst the others followed by the direct mapped implementation. This is because LFU algorithm does not accurately reflect if an entry was used a lot and a direct mapped cache does not employ associativity. Direct mapped caches in most cases perform better than a cache using LFU replacement so in these cases its better not to have a replacement strategy. Although for a 4MB L2 cache, LFU gave a good performance (97%) as cache accesses were more spread out due to size and this helped the counters reflect more accurate values regarding cache line accesses. Still, this is an expensive strategy to implement so it would not be a choice of implementation unless specifically needed.

Generally, Least Recently Used (LRU) strategy was found to be the most effective strategy as every cache access is logged. Random replacement was found to be better for the I-cache running Stress as compared to Least Recently Used (LRU). Random Replacement is always program and time dependent so a degradation in hit rate could be expected for different programs or if the program was run for a longer time. Yet, it is a simple strategy as compared to LRU and could be used to give a reliable performance depending on how the cache would be used. It is also less complex as compared to LRU.

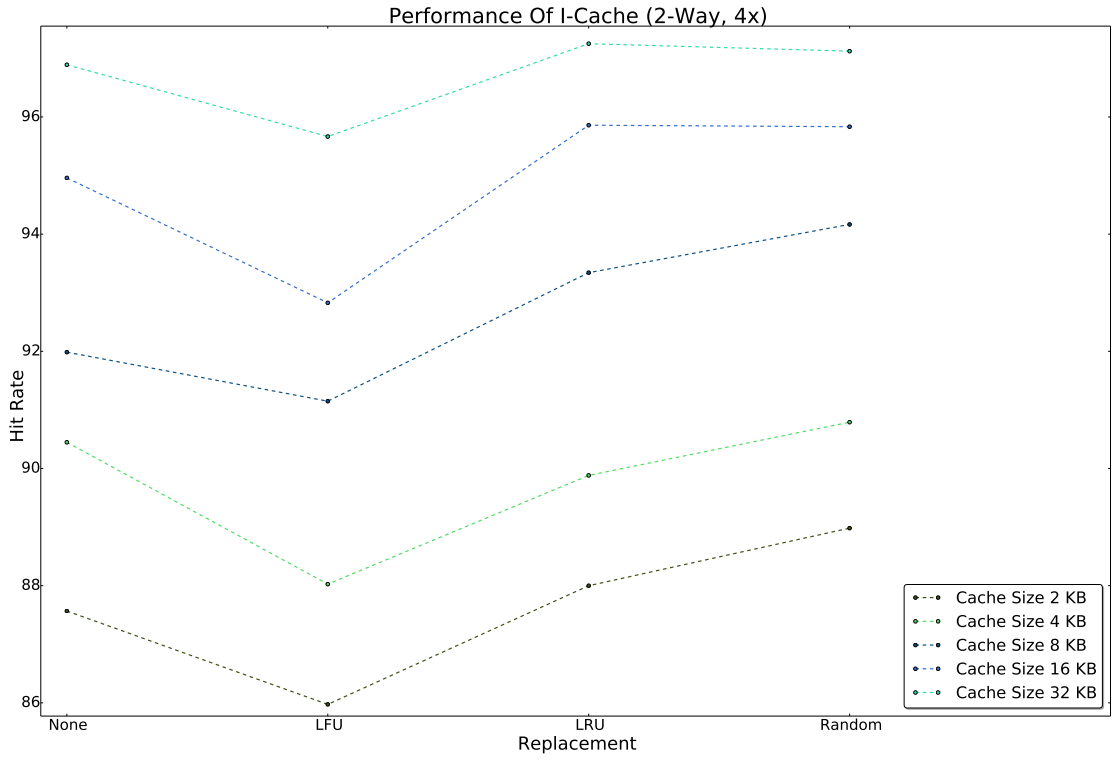


Figure 9.19: Stress test on I-Cache Configurations.

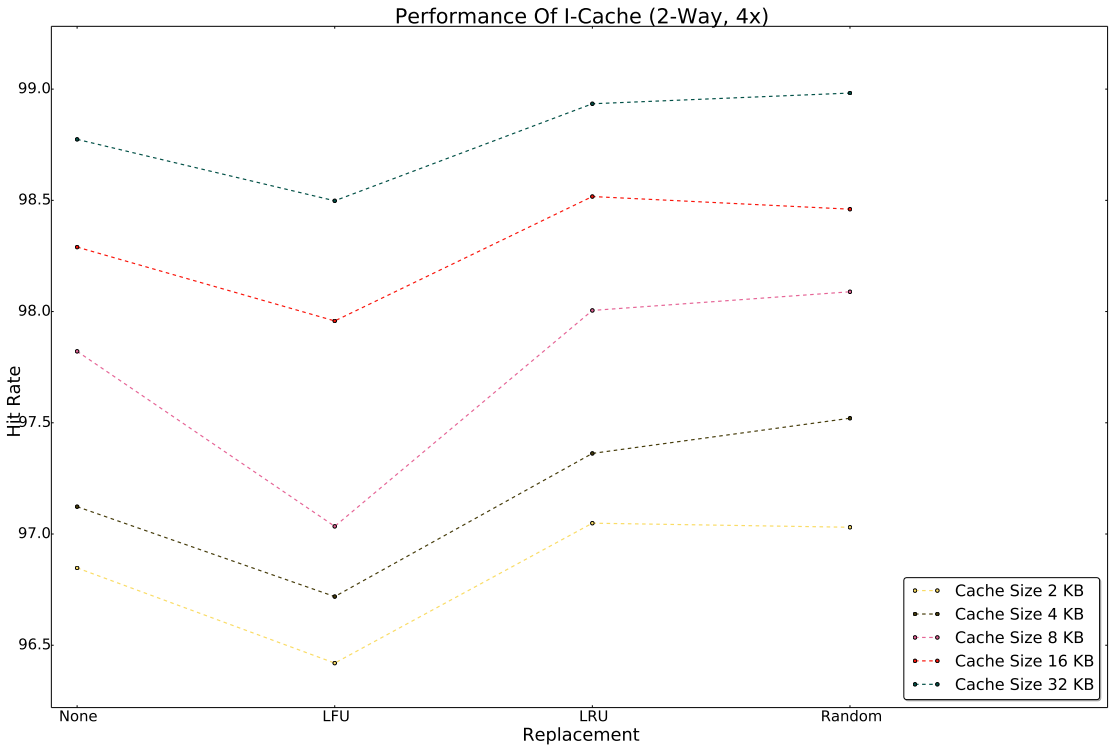


Figure 9.20: SHA test on I-Cache Configurations.

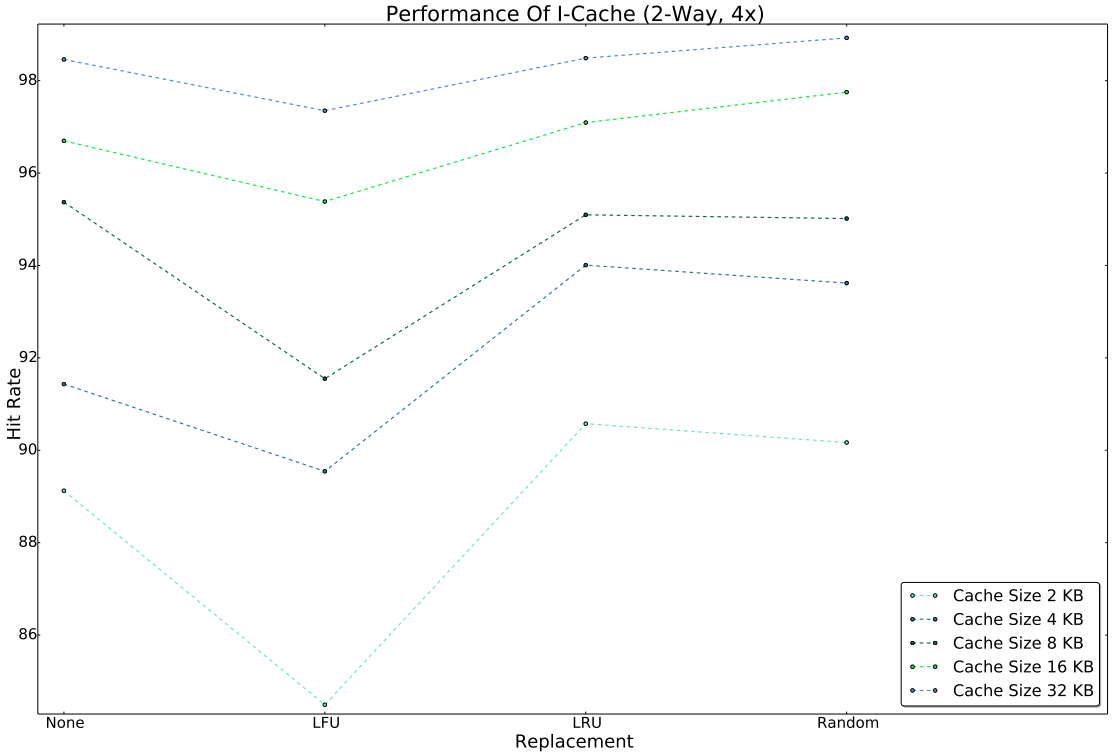


Figure 9.21: 56kB sort on I-Cache Configurations.

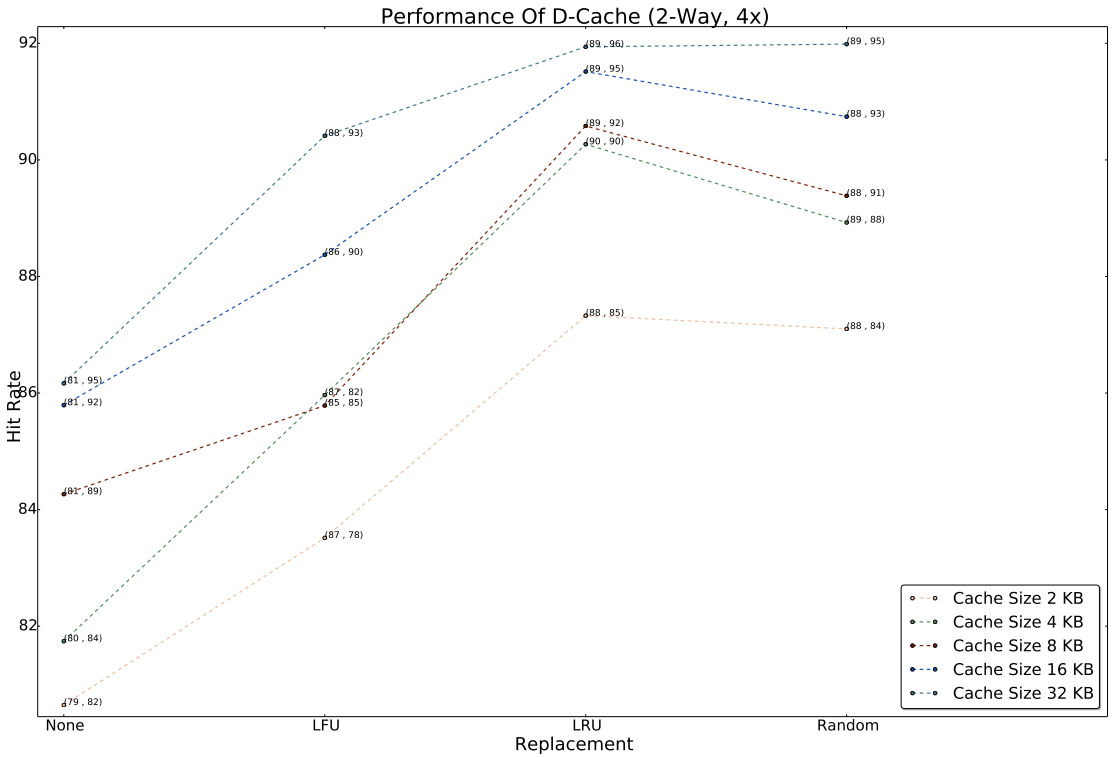


Figure 9.22: Stress test on D-Cache Configurations.

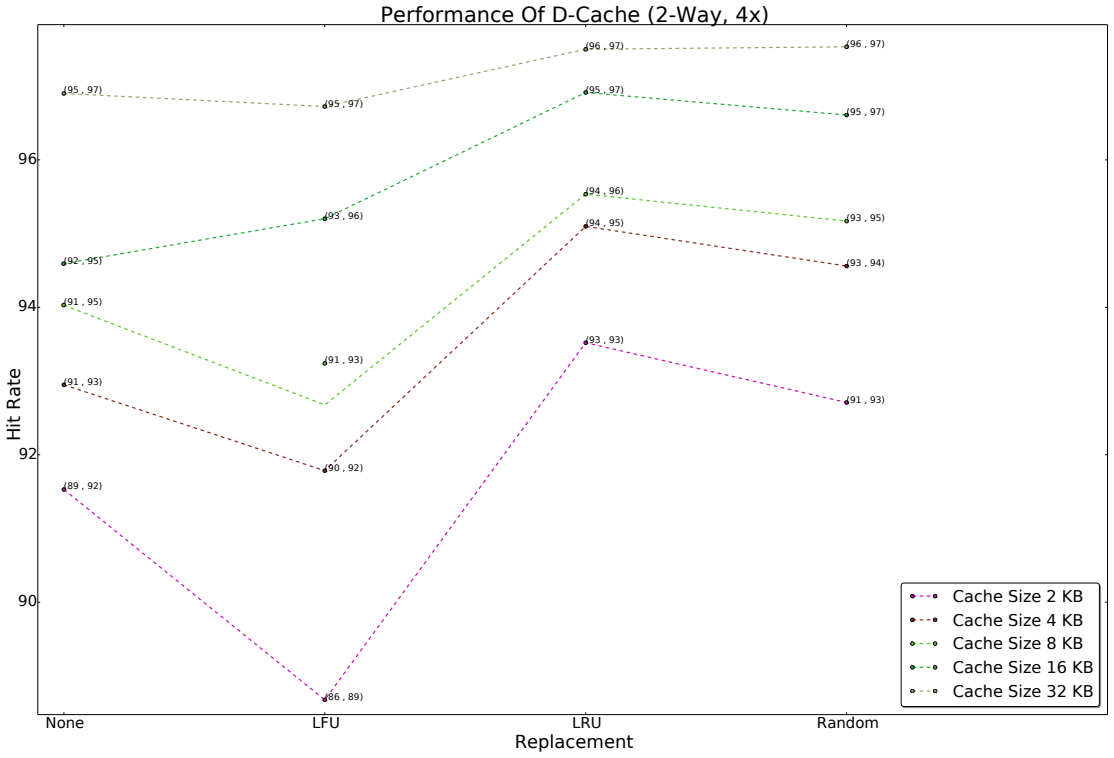


Figure 9.23: SHA test on D-Cache Configurations.

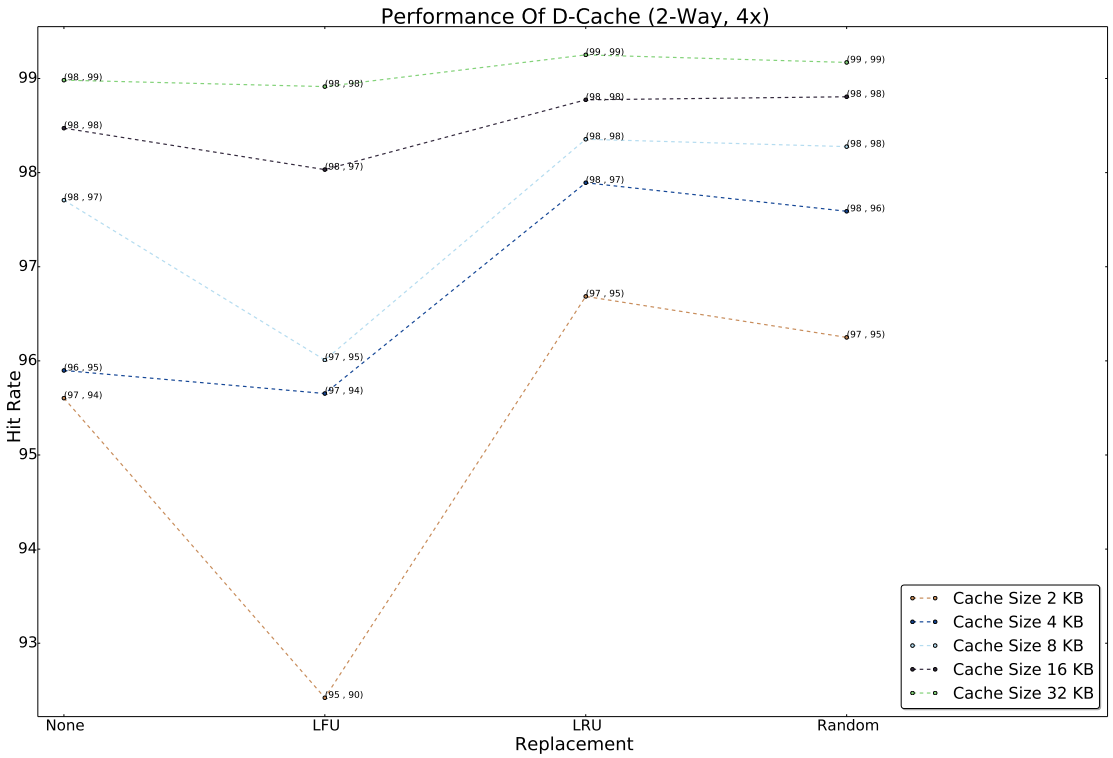


Figure 9.24: 56kB sort on D-Cache Configurations.

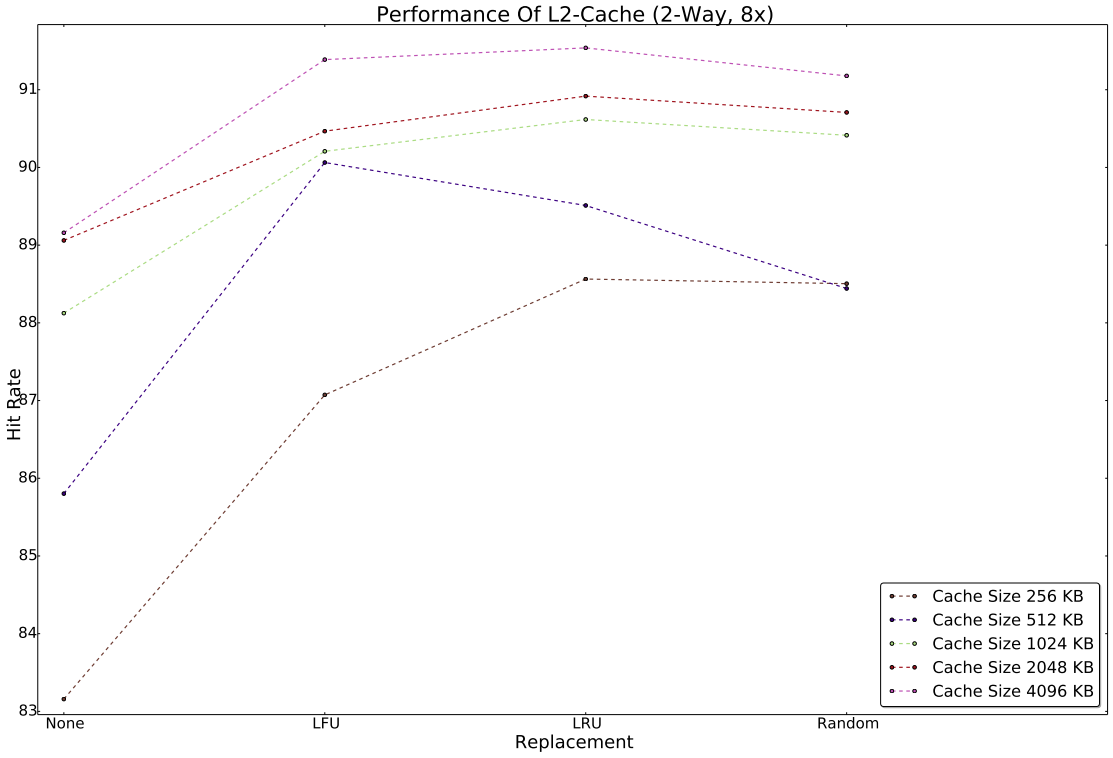


Figure 9.25: Stress test on L2 Cache Configurations.

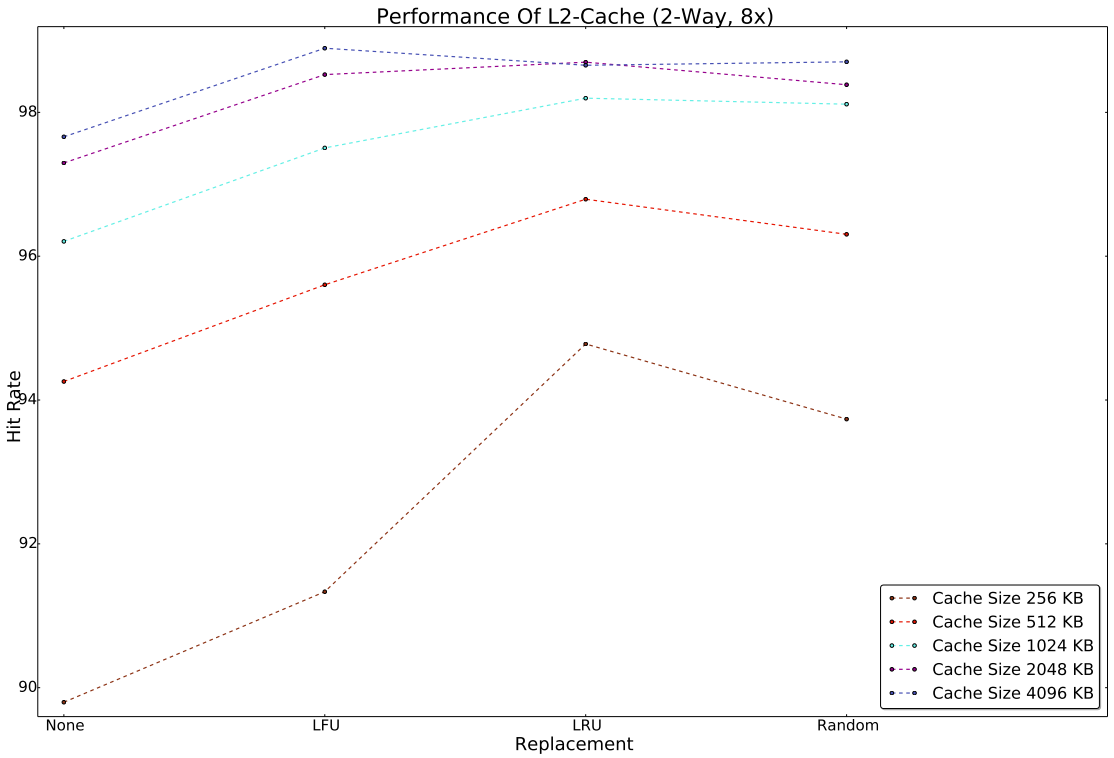


Figure 9.26: SHA test on L2 Cache Configurations.

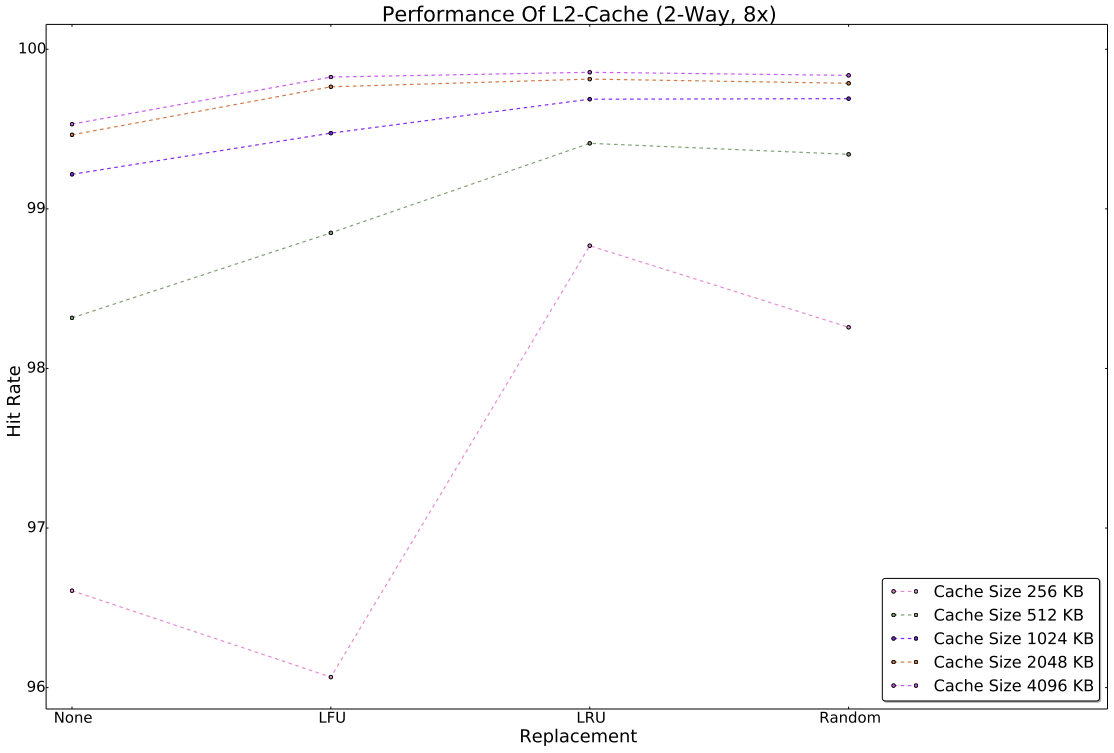


Figure 9.27: 56kB sort on L2-Cache Configurations.

9.5.3 Block Size

Figures 9.28 - 9.36 show the results of using different block sizes for L1 and L2 cache. L1 caches used a block size of 1 to 8 words and L2 caches implemented a block size of 4 to 32 words. 1 word was equivalent to 32 bits. Hit rate increased with an increase in block size. This was expected as a bigger block size meant that a cache line was storing more data and therefore had a greater probability for a hit.

Cache configurations of a one word block size in L1 caches performed the worst as there was no use of spatial locality whereas 8 word blocks performed the best. Similarly for a L2 cache, a block size of 32 words was better than a block size of 4 words though a 4 block word size still gave a hit rate of approximately 90%.

Bigger caches again gave a better performance which was enhanced by bigger block sizes. For the L2 cache, sizes of 2MB and 4MB gave over 98% hit rates for any block size and were the most effective implementations. Big cache and block sizes do give a better hit rate but practically this is expensive and also spatial locality decreases if block size keeps increasing and also the miss penalty associated with the cache increases. Therefore selecting relatively smaller cache and block sizes would be more practical and near similar performances can be achieved as in the case of using a 2MB L2 cache instead of a 4MB one.

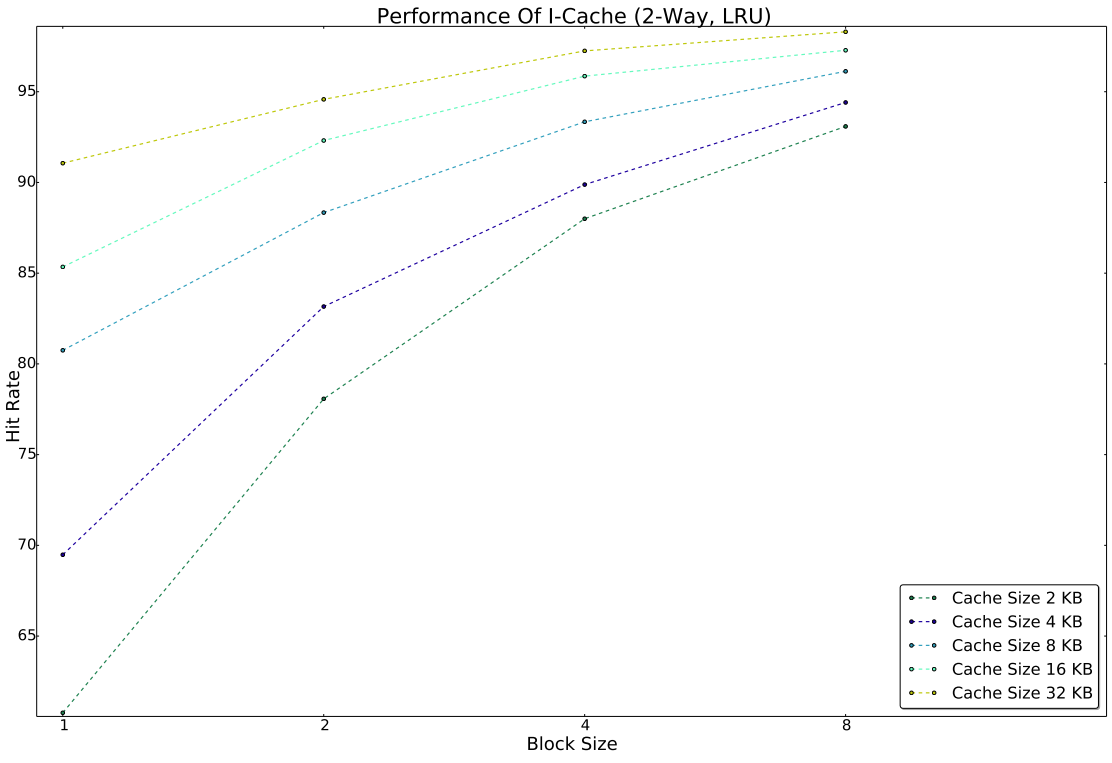


Figure 9.28: Stress test on I-Cache Configurations.

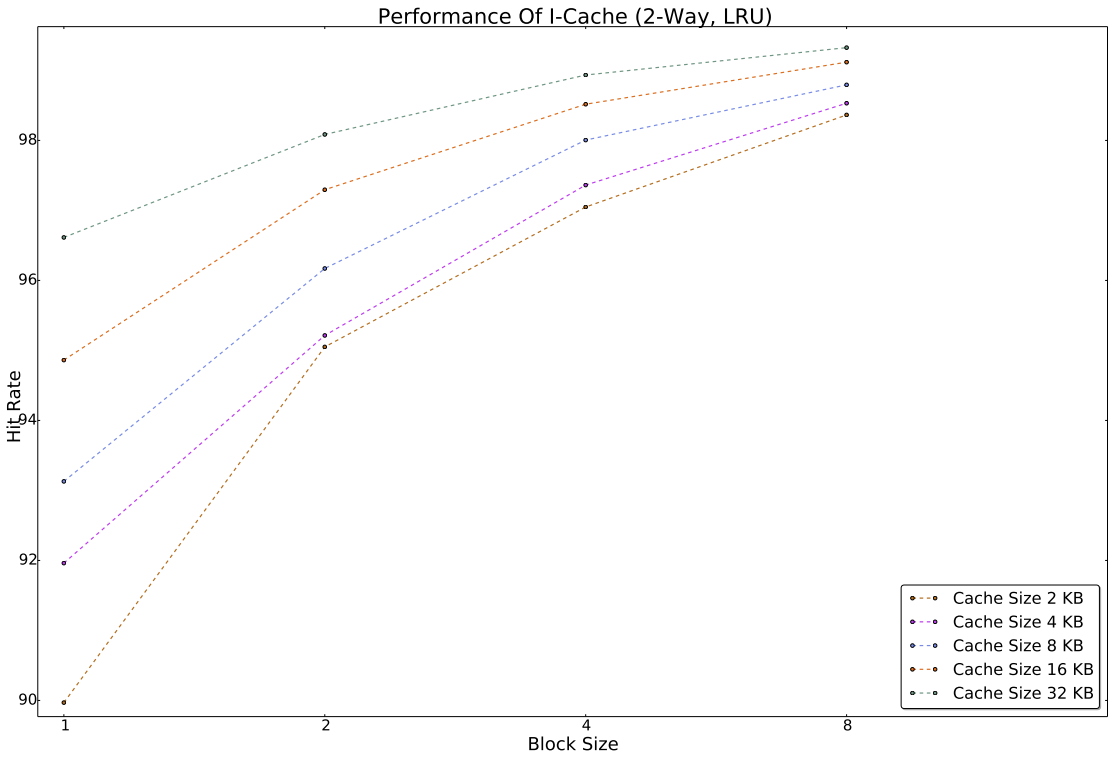


Figure 9.29: SHA test on I-Cache Configurations.

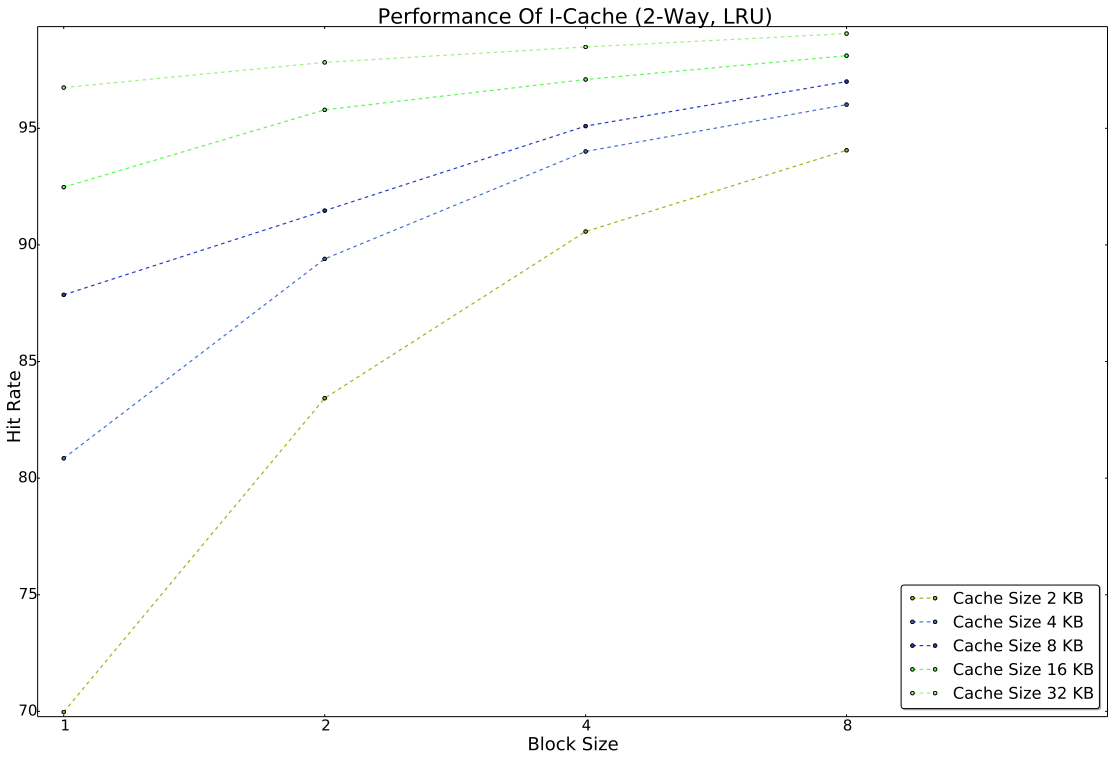


Figure 9.30: 56kB sort on I-Cache Configurations.

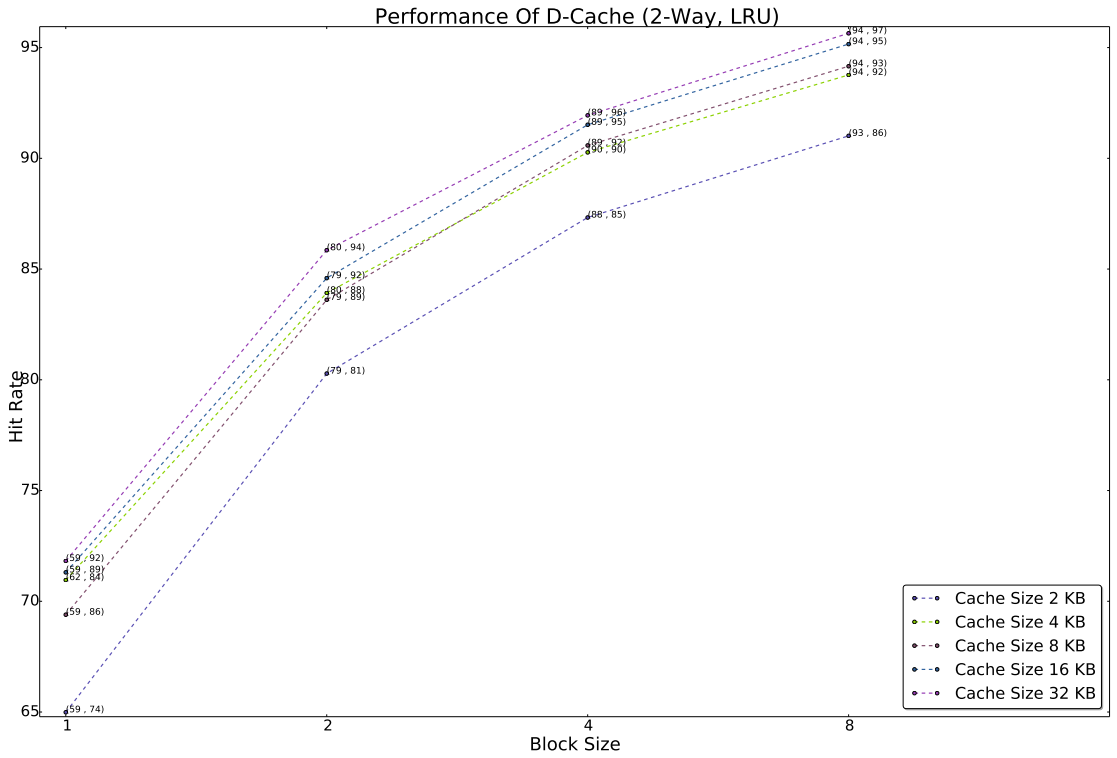


Figure 9.31: Stress test on D-Cache Configurations.

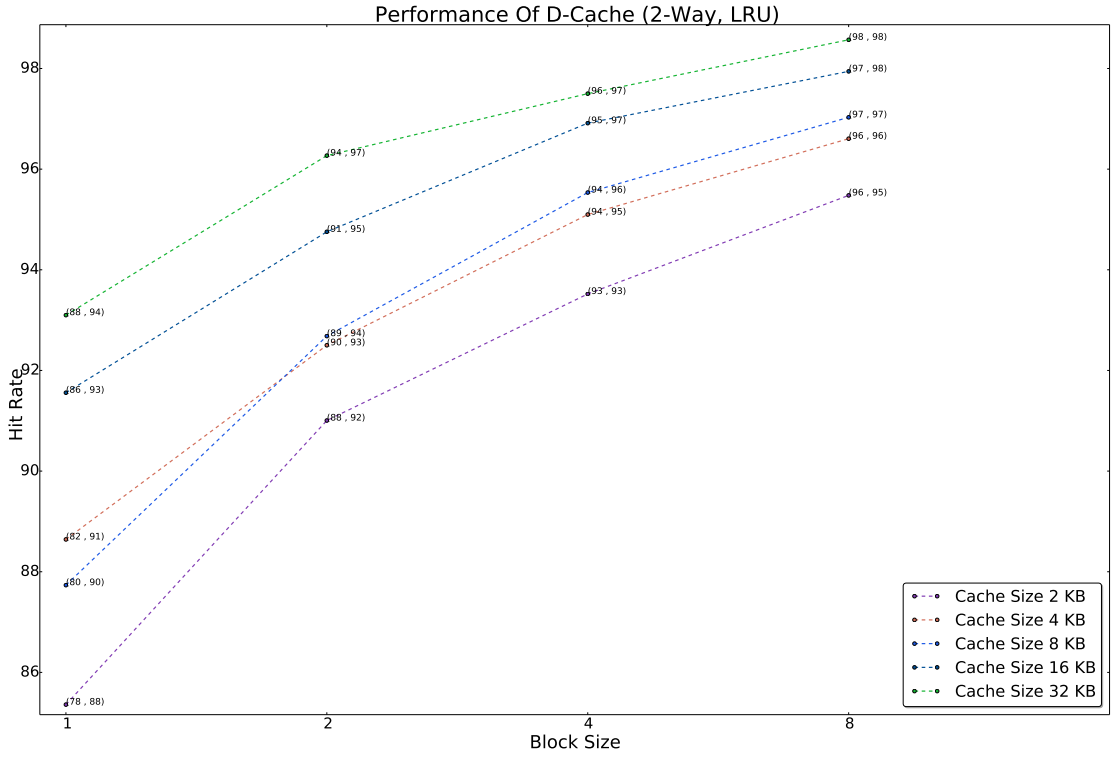


Figure 9.32: SHA test on D-Cache Configurations.

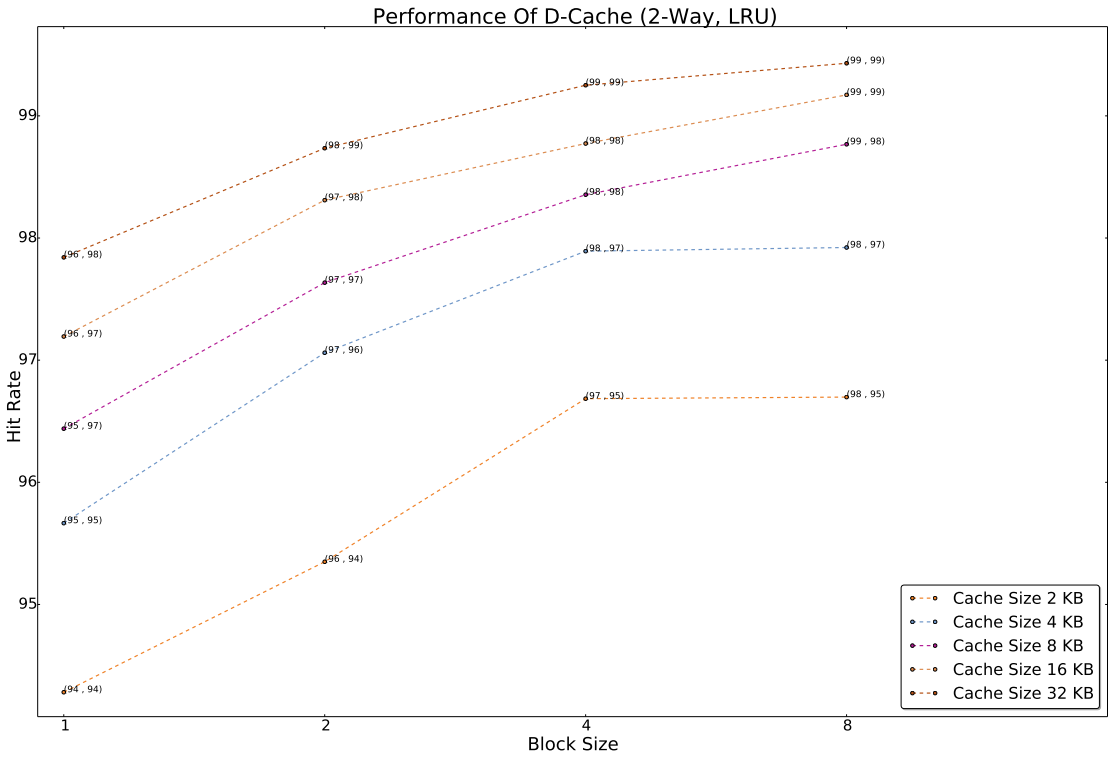


Figure 9.33: 56kB sort on D-Cache Configurations.

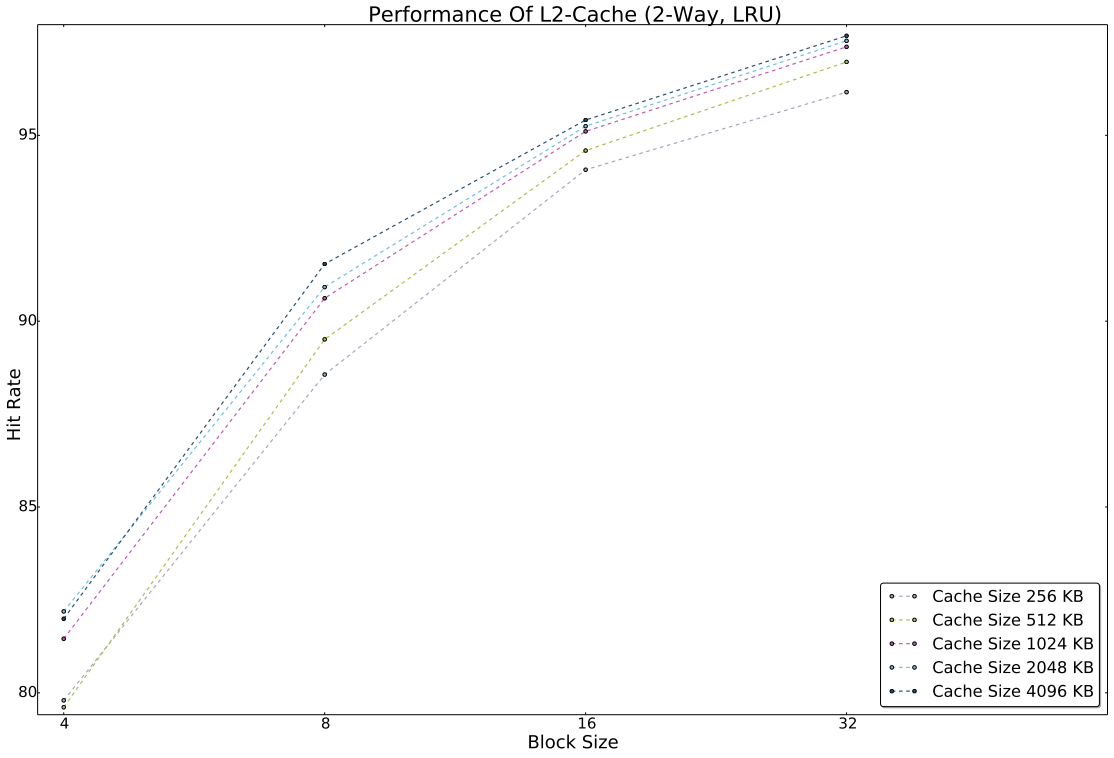


Figure 9.34: Stress test on L2 Cache Configurations.

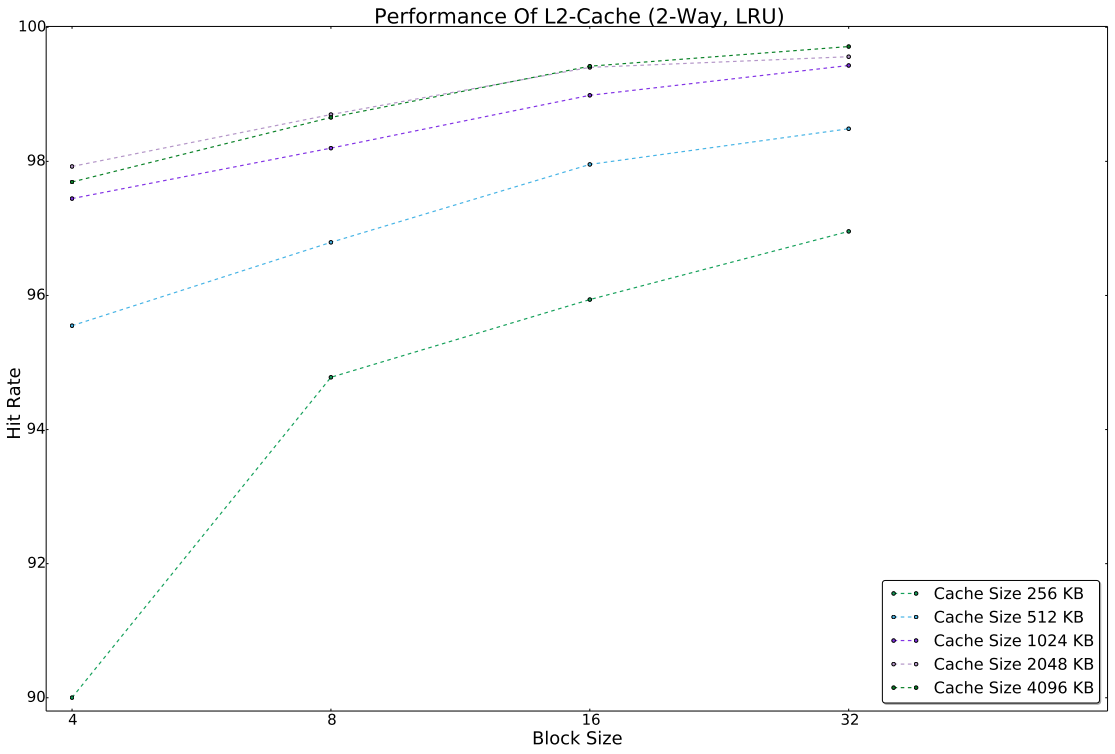


Figure 9.35: SHA test on L2 Cache Configurations.

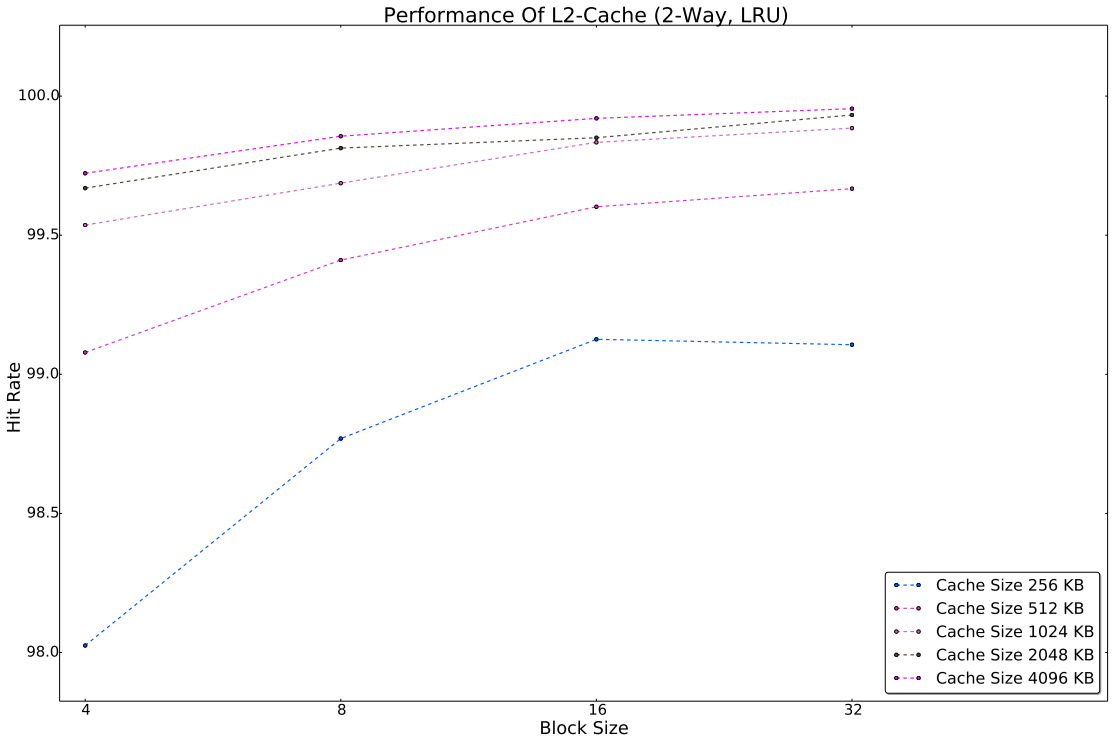


Figure 9.36: 56kB sort on L2-Cache Configurations.

9.5.4 Top I-Cache Designs

The 10 best performing cache configurations are shown in Figures 9.37, 9.38 and 9.39. A four way set associative, 32kB cache with a block size of 8 words clearly performed the best in all cases. Random Replacement was found to be the most effective for the sort program whereas LRU performed better for the other two. Although considering all cases, Random replacement was found to perform better with different cache configurations as opposed to LRU so it can be considered as a default replacement strategy in future designs. The best performers had a hit rate of about 99%. Interestingly, a direct mapped 32kB cache with a 8 word block size gave a hit rate of about 98% and so a simple and yet effective I-cache configuration can be achieved by employing this design as it is less expensive and easier to implement. Even a four way associative, 16kB cache with a block size of 8 words gave a hit rate of 98% - 99% and could be an effective implementation choice for the I-cache.

Therefore, it can be seen that there are multiple design choices for the I-cache. The most effective choice giving the highest hit rate might not always be the most viable and there are many suitable alternatives to choose from that give a near similar performance at reduced cost and complexity.

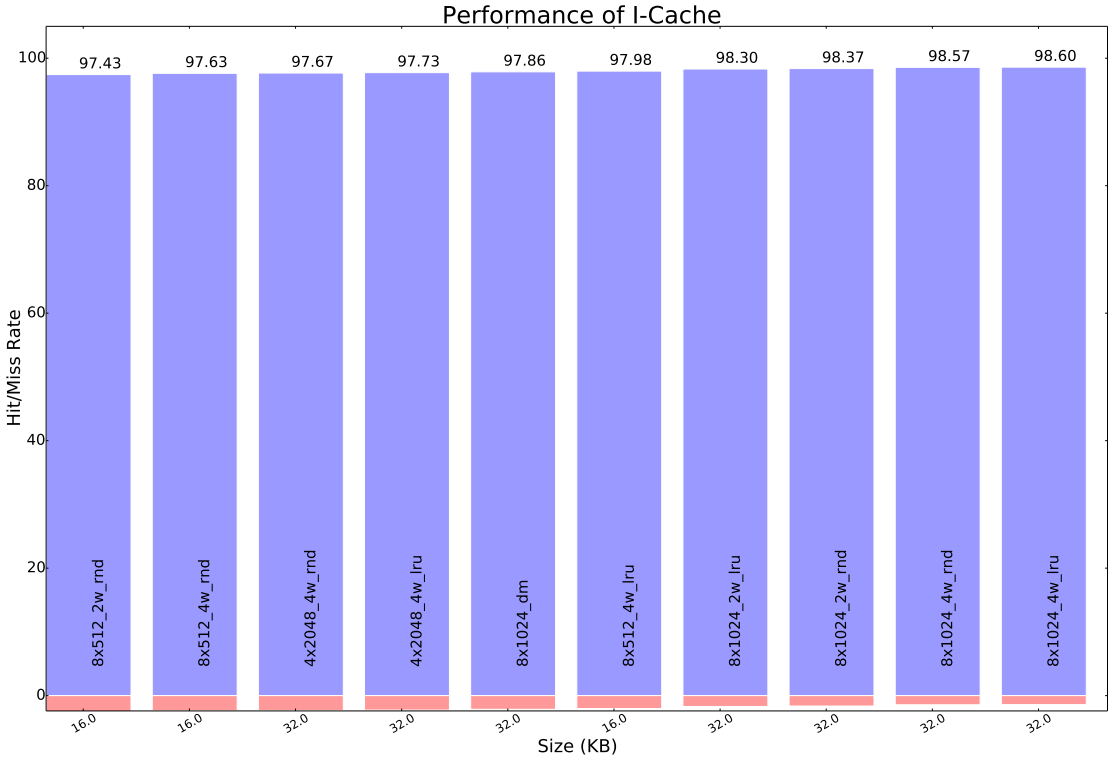


Figure 9.37: Best I-Cache Designs for Stress.

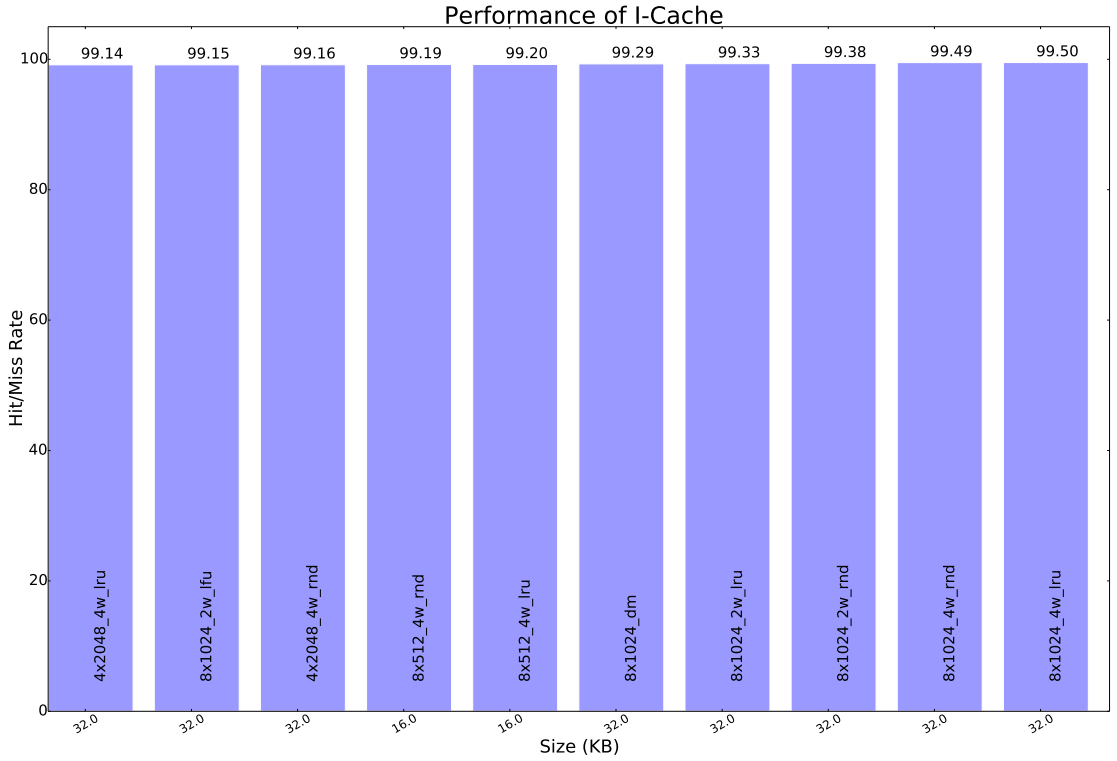


Figure 9.38: Best I-Cache Designs for SHA.

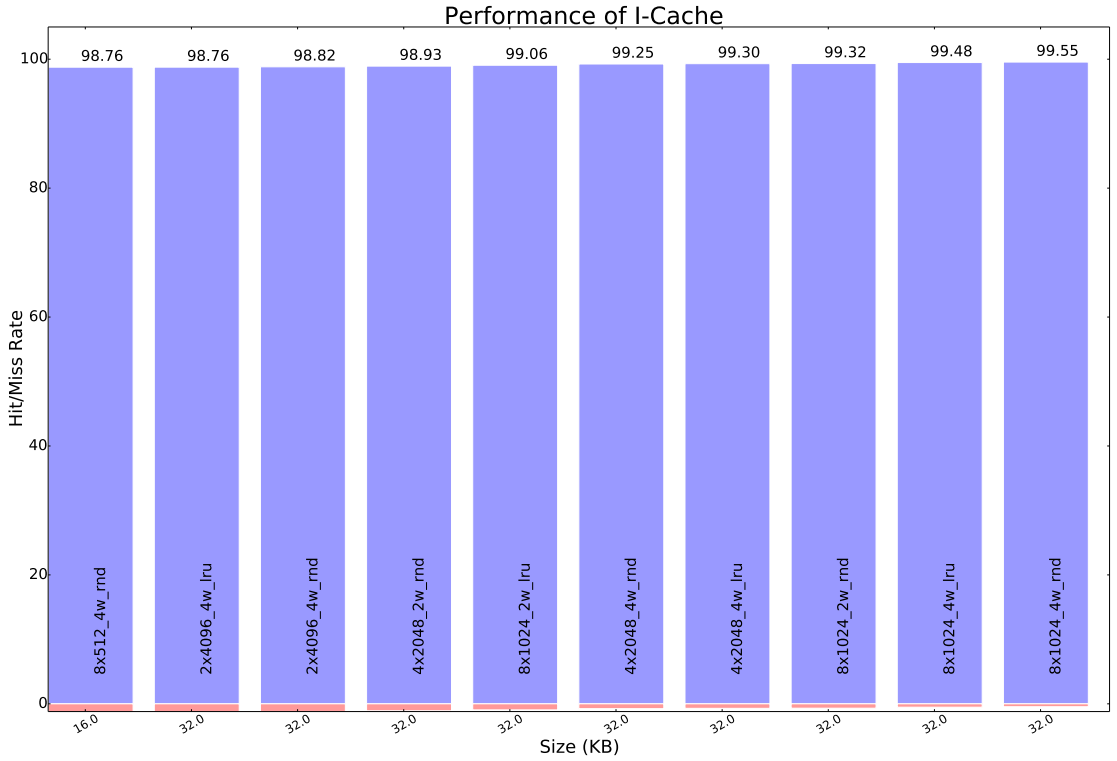


Figure 9.39: Best I-Cache Designs for Sort.

9.5.5 Top D-Cache Designs

From Figures 9.40, 9.41 and 9.42 it can be seen that a 32kB, four set associative cache with an 8 word block size using LRU strategy performed the best amongst the tested cache configurations. A hit rate of approximately 98% was achieved using this configuration. LRU usually performed the best amongst other replacement strategies although the same cache configuration using random replacement gave near similar performance. Therefore, random replacement could be used as a suitable replacement strategy. 16kB caches with the similar configuration also performed extremely well (96%) and a smaller cache size could be viable. Set associative caches performed much better than direct mapped caches. 32kB two way set associative caches performed well and had high hit rates (95%).

A good d-cache implementation could be a four-way associative 16kB cache using random replacement (97 %). There are many other possible permutations to choose from and the choice of configuration is application specific.

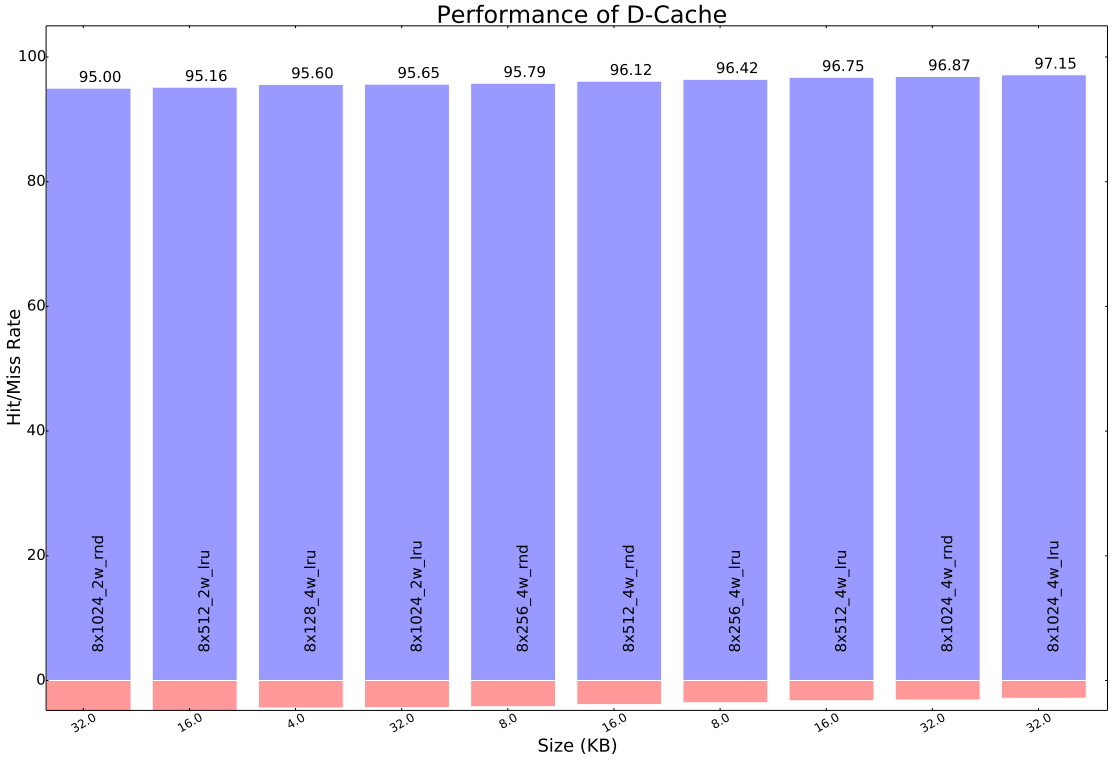


Figure 9.40: Best D-Cache Designs for Stress.

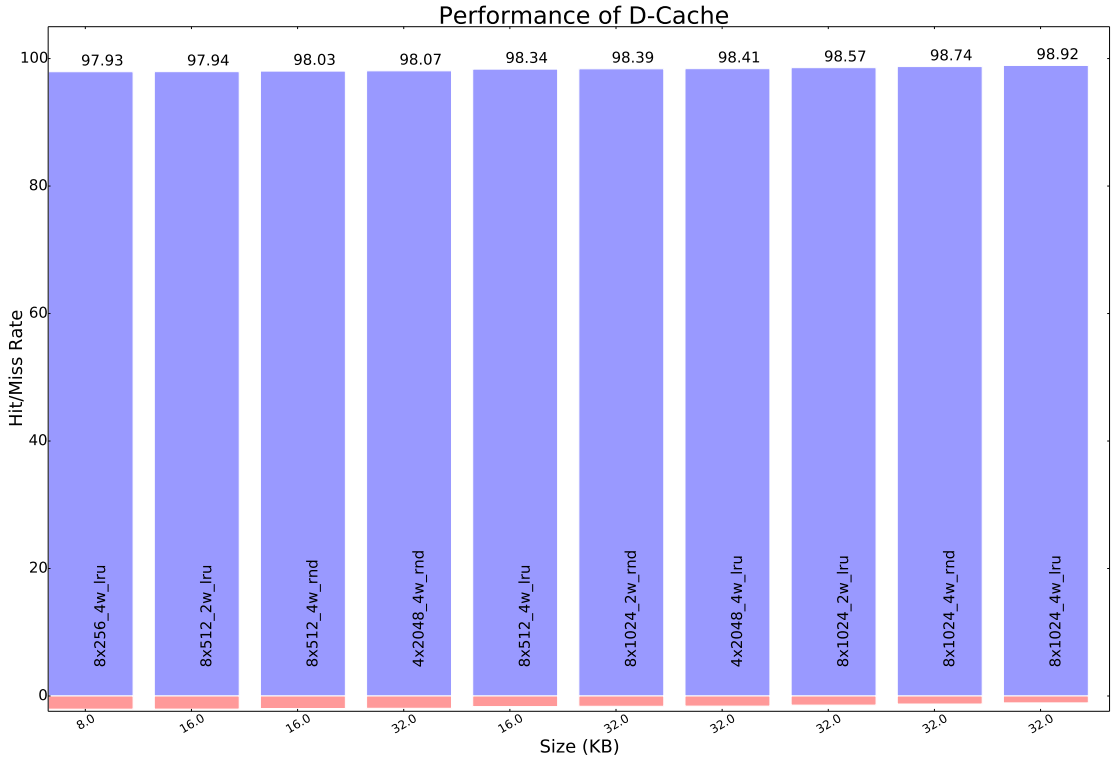


Figure 9.41: Best D-Cache Designs for SHA.

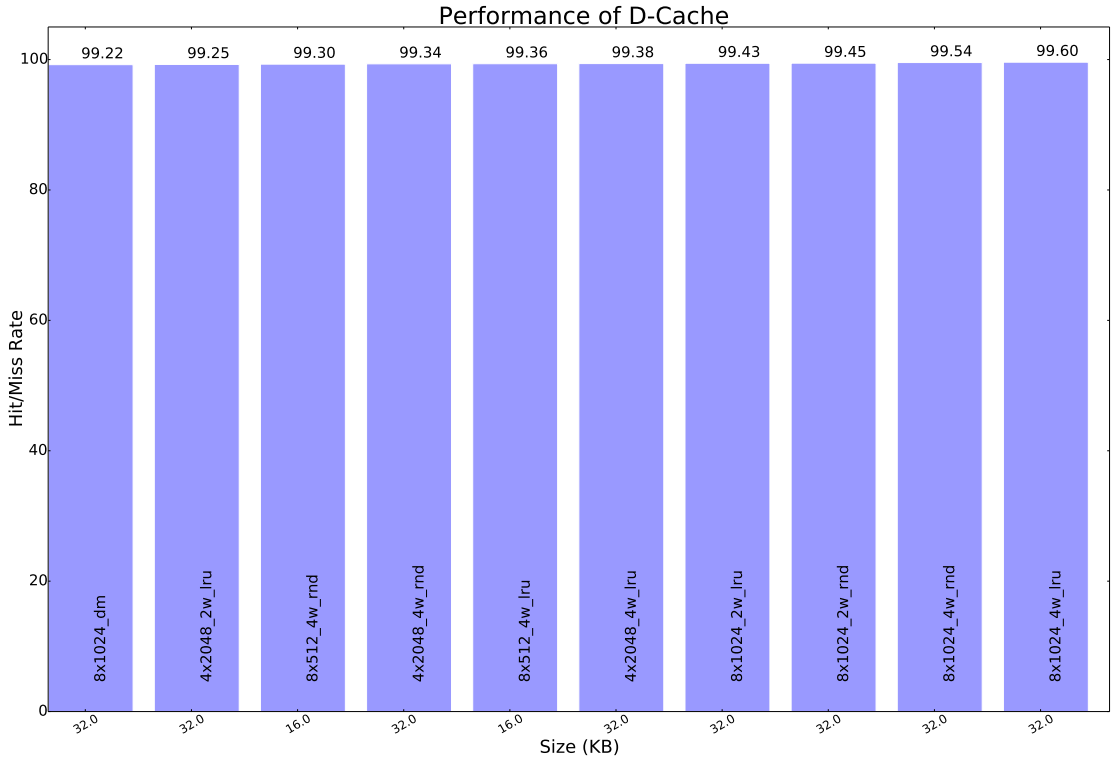


Figure 9.42: Best D-Cache Designs for Sort.

9.5.6 Top L2-Cache Designs

Figures 9.43, 9.44 and 9.45 show that the best configuration for the L2 cache was a 4MB, four way set associative cache with a block size of 32 words using LRU. This was the case for SHA and the sort program and hit rates of 99% were seen. For Stress, a 2MB cache using the same configuration but with random replacement had the highest hit rate (98%). This cache configuration demonstrated a good performance for the other two tests as well. Therefore, it can be seen that a high hit rate can be achieved using a smaller size. Even, a four way set associative 1MB cache using LRU strategy with a 32 word block size performed really well (98%) and could be possible design choice for the L2 cache.

LFU strategy had a good performance as well for certain cache designs as cache size was bigger. Most of the configurations had an average hit rate of 99% and therefore many design choices can be made depending on the application.

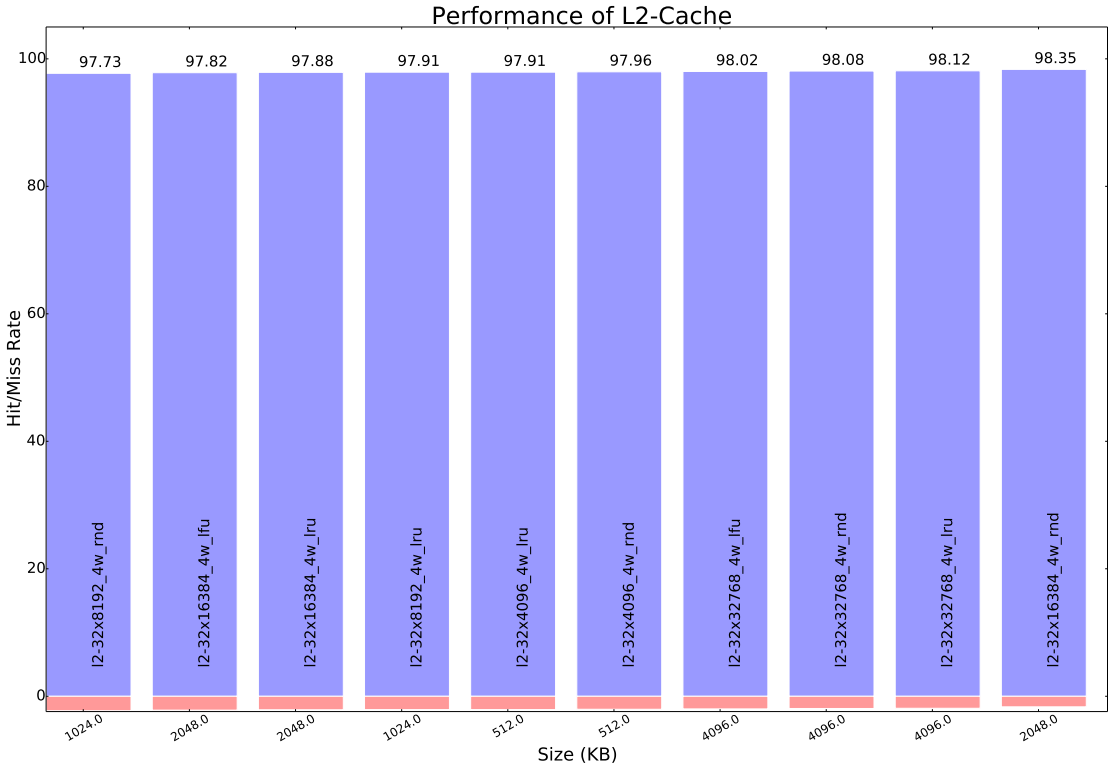


Figure 9.43: Best L2-Cache Designs for Stress.

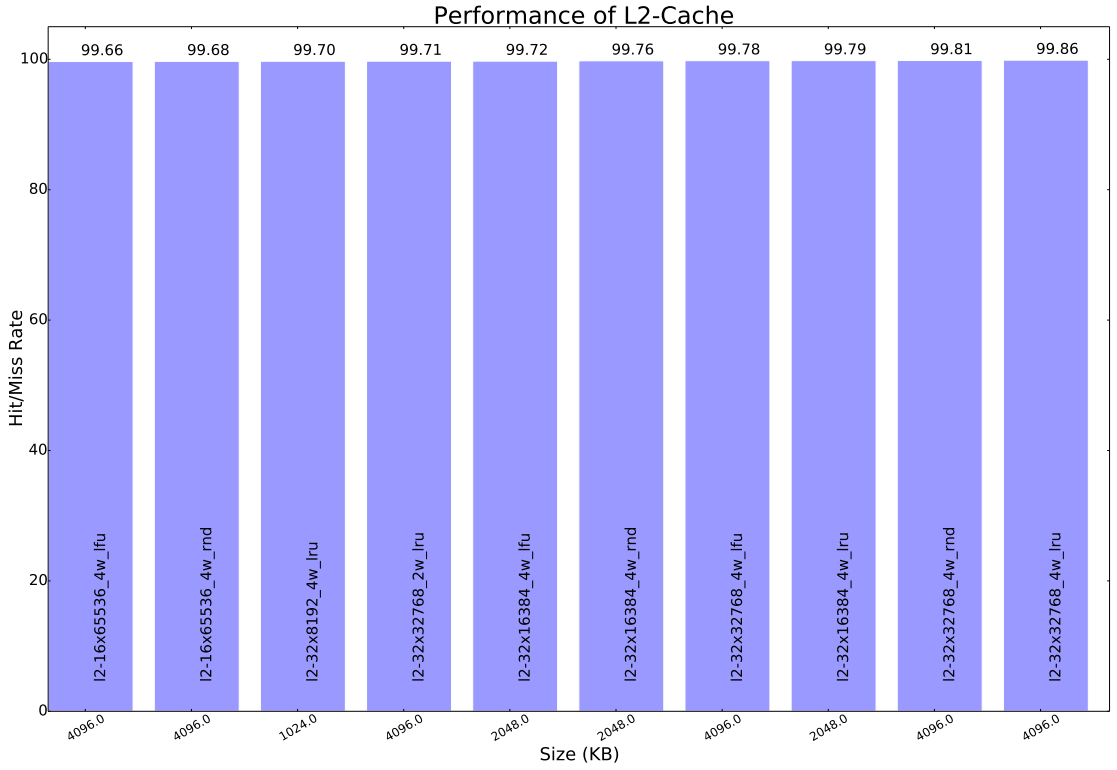


Figure 9.44: Best L2-Cache Designs for SHA.

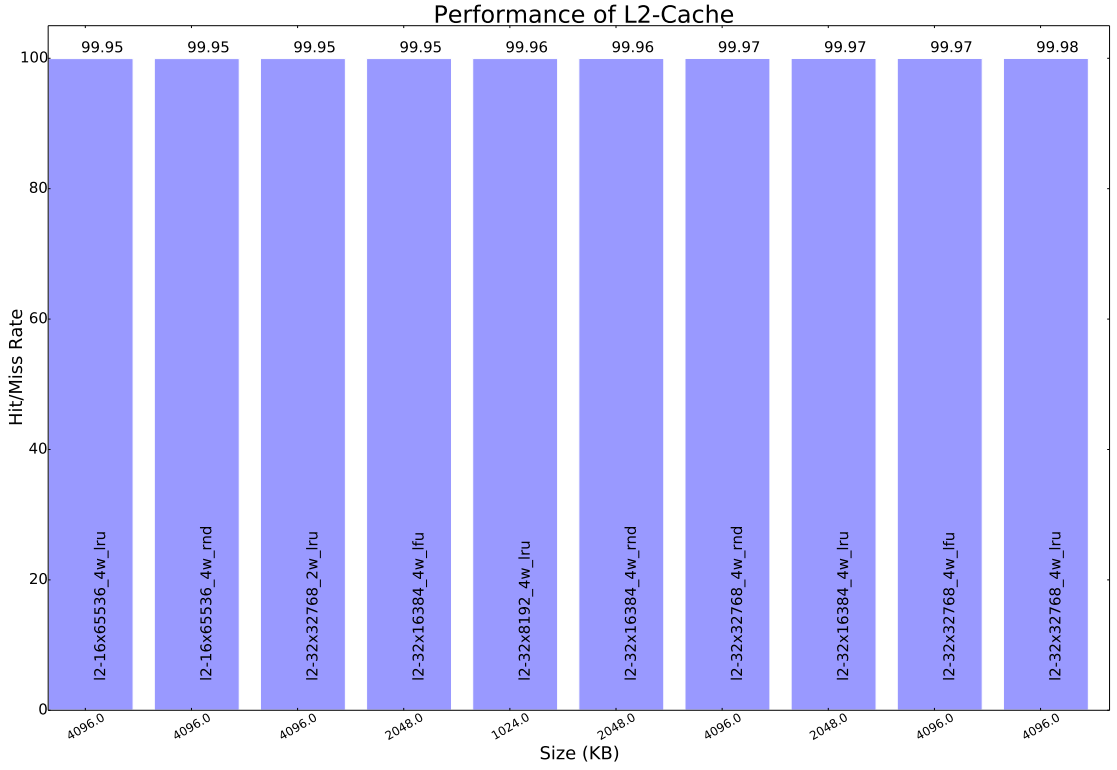


Figure 9.45: Best L2-Cache Designs for Sort.

Chapter 10

Cache Profiling

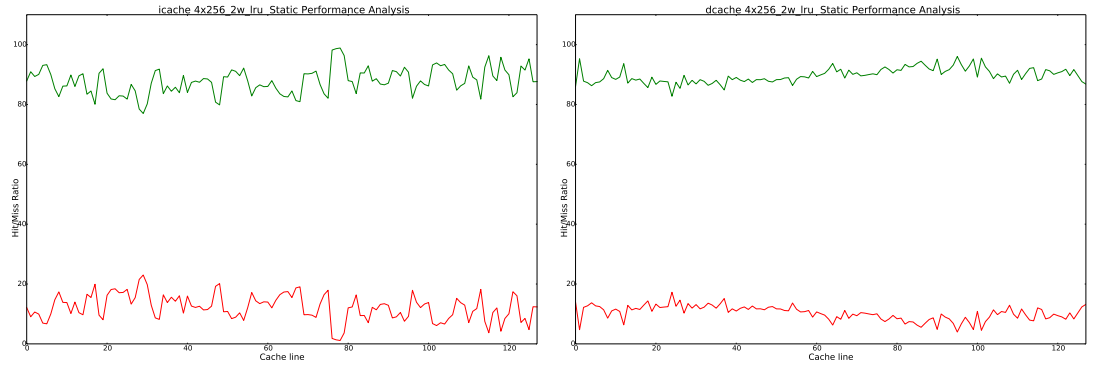
10.1 Offline (Static) Cache Profiling

Offline cache profiling shows hit and miss ratio for each line in a cache. It is performed in this project using *QEMU-off-line-Profiling.py* Python script. Firstly, it asks a user for a full path to a log file and checks if the file exists. The file containing hit and miss counts for each cache line is opened and the data is placed in corresponding lists in Python code. Five lists are used for L1 data cache and three lists for L1 instruction and L2 unified cache (see Figure 9.1(b) for details).

As described in Chapter 9 log files with hit/miss counters have been generated for a number of programs. This section presents hit/miss ratio patterns for each of the programs together with the command used to execute them. All figures use 4kB 2-way set-associative L1 caches with 4 words in a cache line and least recently used replacement algorithm.

10.1.1 Stress memory for 10s – *stress*

Command used: `stress --cpu 1 --io 1 --vm 10 --vm-bytes 8M --timeout 10s`.

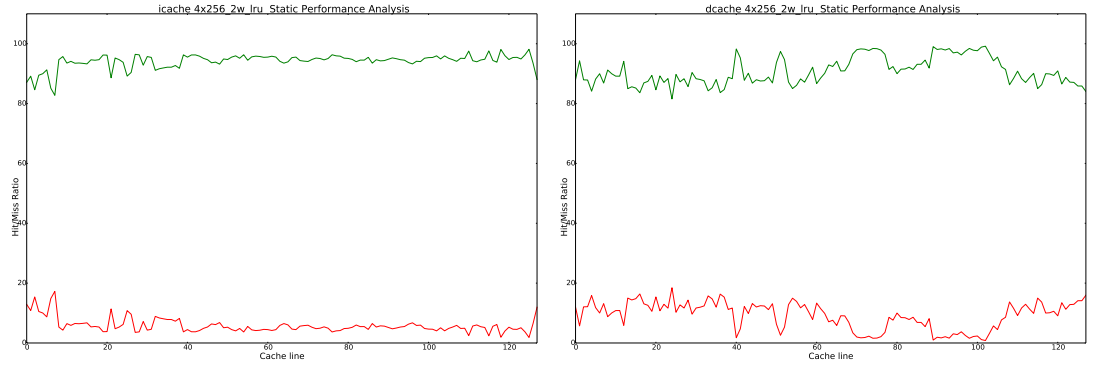


(a) I-cache (b) D-cache

Figure 10.1: Stress program running for 10s – cache profiling

10.1.2 Create and copy 512kB of random data – *dd*

Command used: `dd if=/dev/urandom of=/dev/zero bs=524288 count=1`.

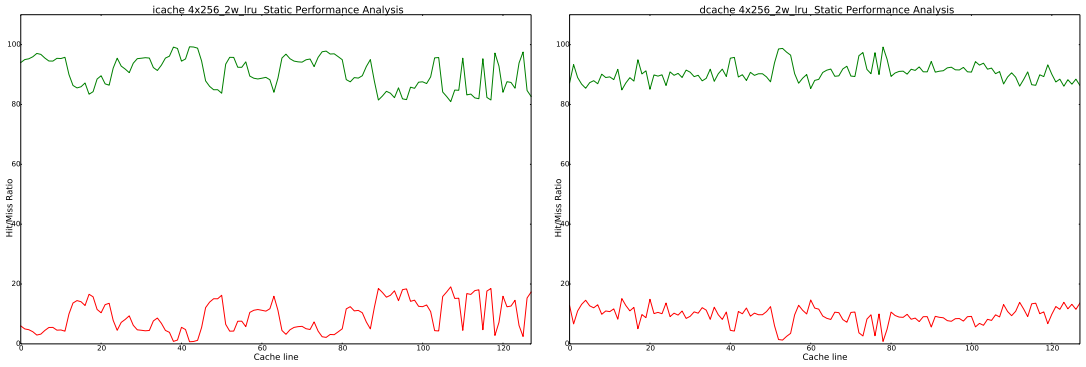


(a) I-cache (b) D-cache

Figure 10.2: Dd program copying 512kB from /dev/urandom – cache profiling

10.1.3 Extract 128kB file – *bunzip2*

Command used: `bunzip2 -d /testdir/bztest.bz2`.

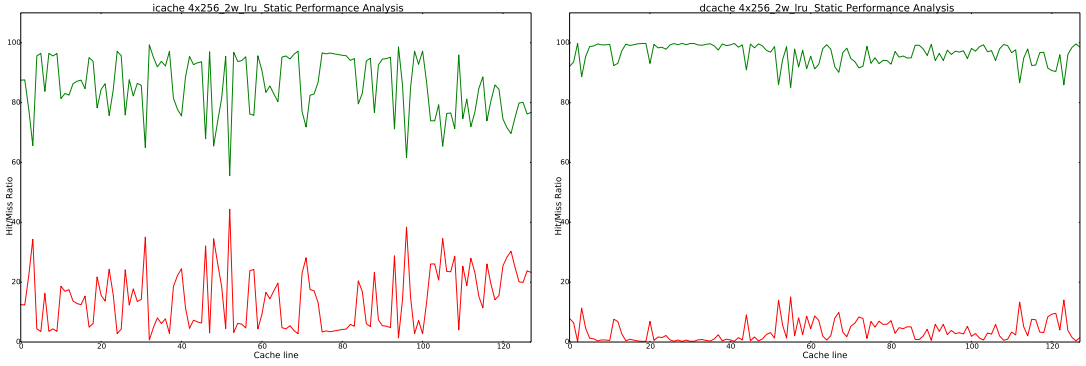


(a) I-cache (b) D-cache

Figure 10.3: Bunzip2 program extracting 128kB file – cache profiling

10.1.4 Search for ”if” text in multiple files – *grep*

Commands used: `grep -r "if" /lib32/` and `grep -r "if" /lib/`.

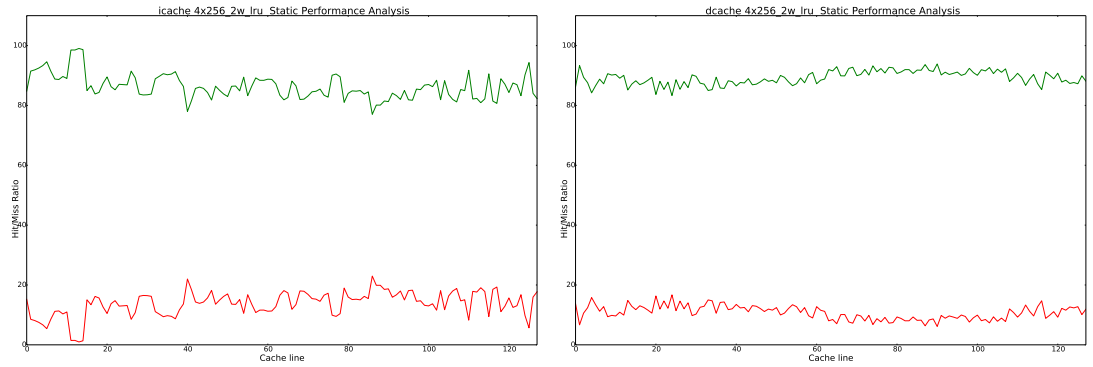


(a) I-cache (b) D-cache

Figure 10.4: Grep program searching for ”if” – cache profiling

10.1.5 Compare files in two directories – *diff*

Command used: `diff -r /testdir/difftest /testdir/difftest2`.

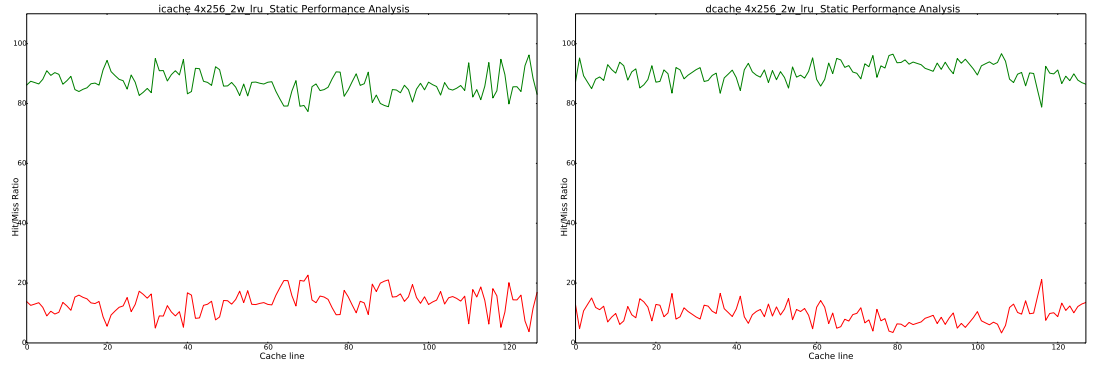


(a) I-cache (b) D-cache

Figure 10.5: Diff program comparing directories – cache profiling

10.1.6 Find file name or folder name – *find*

Command used: `find / -name "lib" 2>/dev/null`.

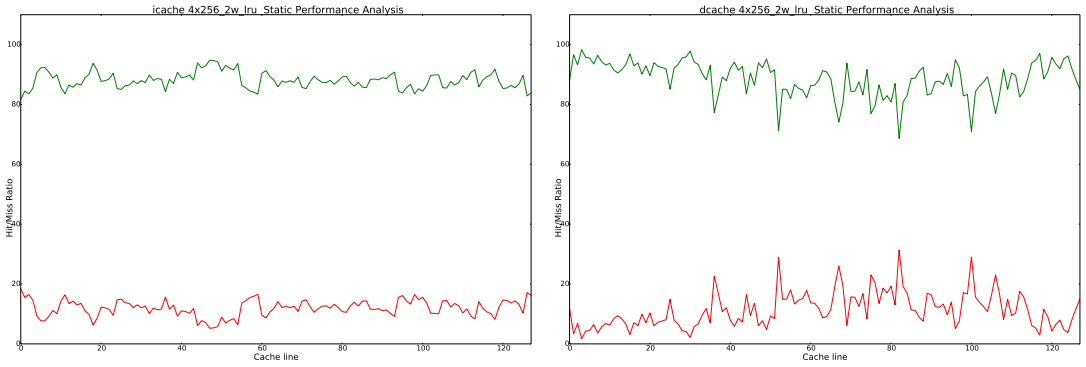


(a) I-cache (b) D-cache

Figure 10.6: Find program looking for "bin" – cache profiling

10.1.7 Chess game – *gnuchess*

Command used: `gnuchess < /home/default/chessgame`.

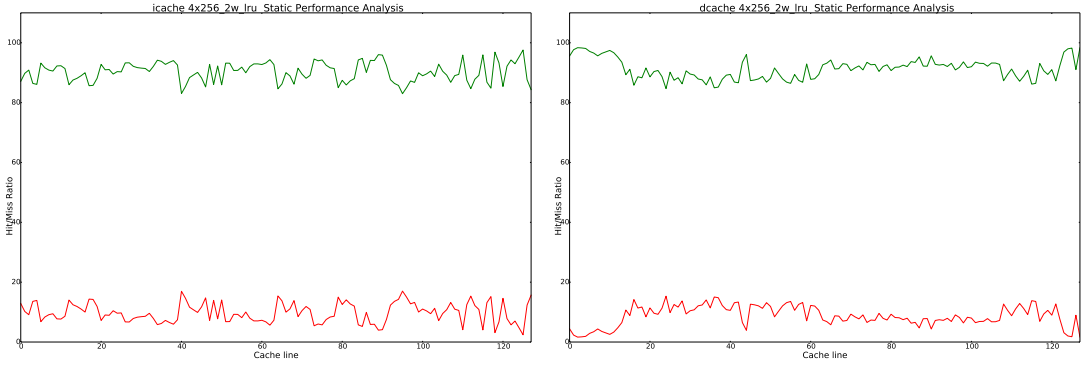


(a) I-cache (b) D-cache

Figure 10.7: Computer playing chess with itself – cache profiling

10.1.8 Sort data in 56KB file – *sort*

Command used: `sort -g -r -k 2 /testdir/sorttest56K -o /testdir/sorted`.

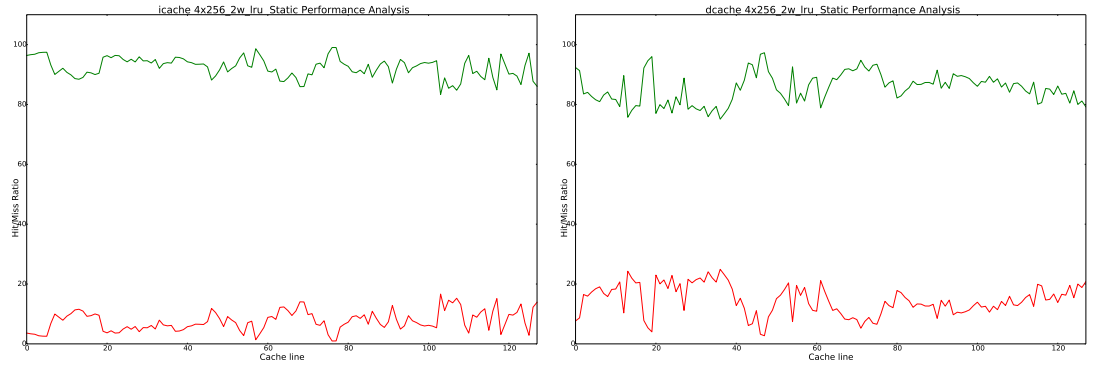


(a) I-cache (b) D-cache

Figure 10.8: Sort program sorting 56KB cache log file – cache profiling

10.1.9 Convert 300KB bmp image to png – *convert*

Command used: `convert /testdir/testimg.bmp /testdir/outimg.png`.

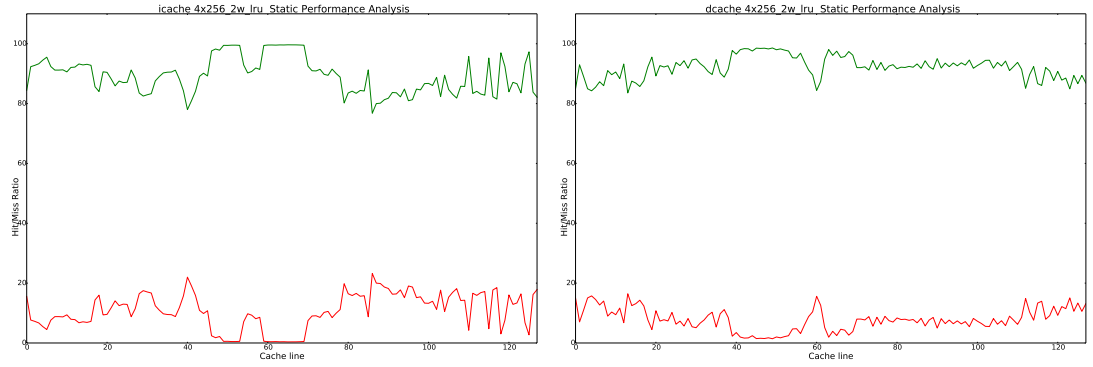


(a) I-cache (b) D-cache

Figure 10.9: Imagemagick program converting bmp to png – cache profiling

10.1.10 Compute 256-bit checksum of 10MB file – *sha256sum*

Command used: `sha256sum /testdir/testfile10M.bz2`.

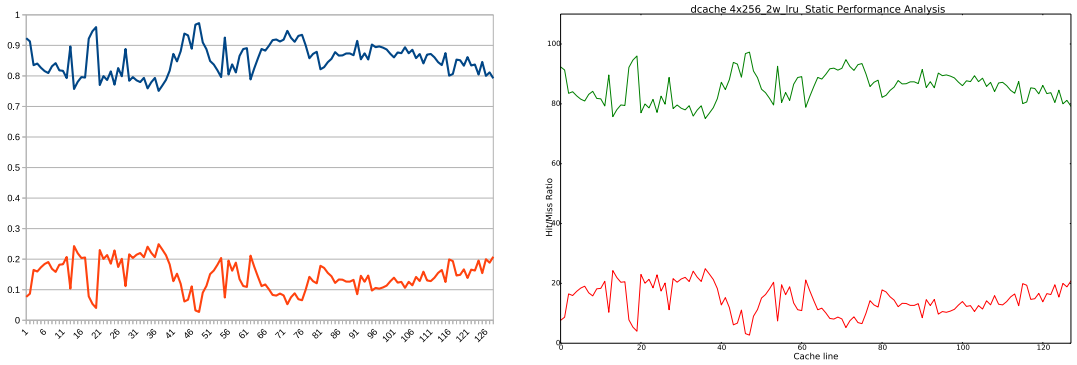


(a) I-cache (b) D-cache

Figure 10.10: Sha256sum program calculating checksum of 10MB file – cache profiling

10.1.11 Comparing Python plot with LibreOffice Calc plot

To verify that Python script draws correct plots, the same data is opened in LibreOffice Calc and the figures are compared. Figure 10.11(a) shows a plot from Calc and Figure 10.11(b) the corresponding plot created using Python script. They are basically the same, which confirms that the script works properly.



(a) LibreOffice Calc

(b) Python script

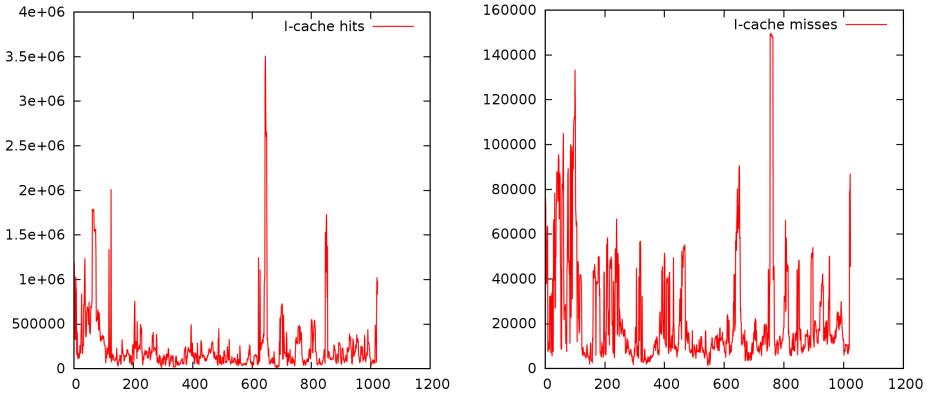
Figure 10.11: Comparison of LibreOffice Calc and Python plots

10.2 Online Analysis – Live Display

As described in the User Guide (Chapter 5), live data display is done using Gnuplot and can be turned on by passing *gnuplot_cache* option to either QEMU system or user mode. This produced a maximum of 6 separate plots – instruction cache hit and miss count, data cache store hit and miss count and data cache load hit and miss count. By default, the y range is set to 1000000 but autoscale can be turned on using a button on a Gnuplot graph window. The x range automatically adjusts to a cache size. Figures 10.12(a) - 10.14(b) show all possible live graphs with sample plots. These are generated by booting Linux in system mode and executing a number of standard Linux commands. The following cache options have been passed to QEMU:

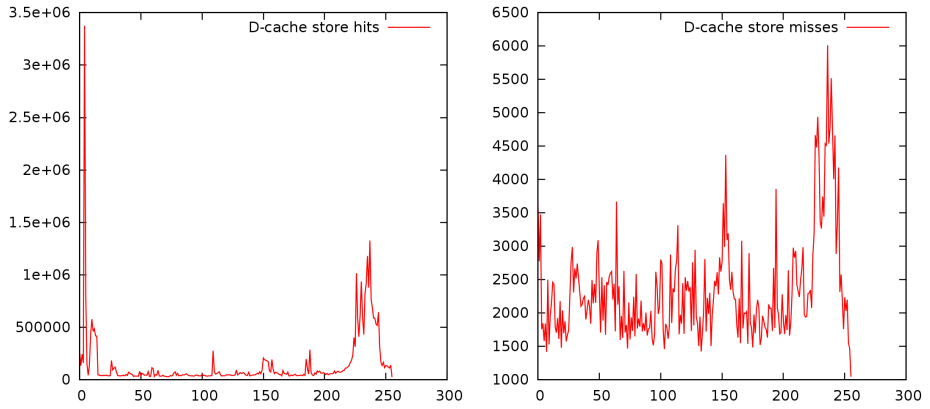
```
$ -icache 4x1024_dm -dcache 8x1024_4w_lru -gnuplot_cache hmstlo
```

Note that autoscale has been used and each plot has different y range.



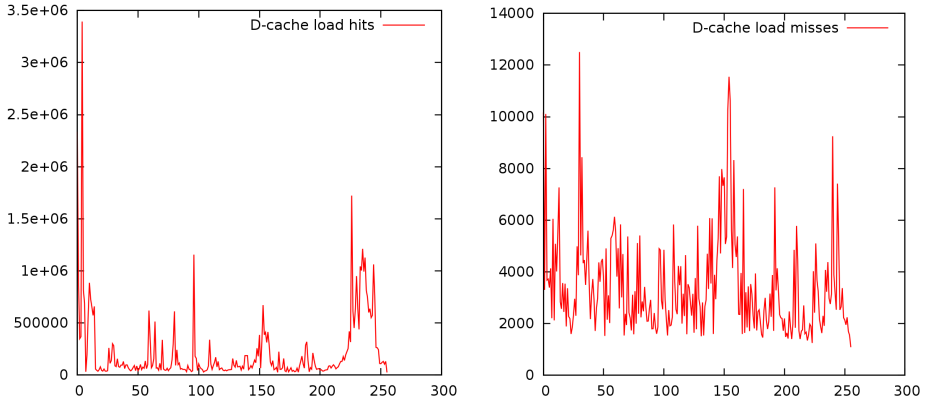
(a) hit count (b) miss count

Figure 10.12: I-cache – live display



(a) hit count (b) miss count

Figure 10.13: D-cache, store instruction – live display



(a) hit count (b) miss count

Figure 10.14: D-cache, load instruction – live display

Chapter 11

Project Management

This chapter outlines the group dynamics, and the methods used to ensure efficient progress and cooperation.

11.1 Initial Meetings

In the first few group meetings, once the project specification outline had been received, the primary goals were to get the specification clarified, and to decide how best to make use of the skills present in the group. As all of the group members had more background in hardware design than software, this was a fairly interesting stage of the project. It was interesting to see how best each person's skills could be best applied to a project that was definitely software based, but required some hardware knowledge. In particular, two of the group members, Ricardo and Rafal, are more experienced in Linux and software development than the other members, while Aditya and Nayaab are experience in computer architecture, and David has skills in Python coding and team management (see Figure 11.1). It was realised that this was actually a very good combination of skills to have, and that the project would be a good opportunity to develop software development skills, while working on something that is intrinsically related to hardware.

In addition, it was very helpful to meet with the industrial contact, Andrew Bennett, in the first two weeks of the project. At this meeting, all the major doubts in the specification were cleared up, and a fairly good picture of the final goal was presented. This greatly increased enthusiasm in the project.

11.2 Allocation of Tasks

There was some overlap in skills between the group members, which proved useful as some parts of the project were better handled by more than one person. In addition,

Figure 11.1: Spider diagram showing approximate skills distribution in the group

this make it simpler to document the project, as the knowledge of each part was shared around the group. During the first stages of the project, the work was much more spread out, in order to cover as much ground as possible. Rafal worked purely on the kernel building and configuration, Ricardo investigated the structure of QEMU, Aditya and Nayaab investigated cache models, and David focussed on cache profiling methods. Then, over a few weeks, these separate parts were brought together into a coherent product. This worked well, as by week 3, each person was an expert in one particular area important to the project.

David Mahmoodi was unanimously selected to be the team leader, whose primary tasks were to communicate with the client and supervisor on behalf of the team, and coordinate the groups work effectively.

11.3 Gantt Chart

Once the specification questions had been answered, it was possible to properly allocate skills and come up with a Gantt chart that represented the intended deadlines for different goals. This original Gantt chart is shown in Appendix B. By the first presentation, 5 weeks into the project, we were a week or two ahead of our expectations in some areas, and revised our goals to compensate. By the final presentation however, we were approximately where we expected to be – just at the last major deadline, and ready to write the report.

11.4 Maintaining Productivity

It was important to keep the group productive throughout the project, in order to be successful. To this end, a 'group' was created on the Facebook social media website, which made communicating between group members very easy. On this group, all project related questions and non-code documents (e.g., meeting minutes) were posted, so that everyone had access to them. In general, this resulted in a very good work ethic from start to finish. In addition, a large portion of work was carried out in the GDP lab space allocated to the group, which allowed face-to-face communication between members.

Regular meetings were held with the project supervisor, and even more meetings were held between group members, where the project progress and remaining goals were discussed. This included taking minutes, and setting weekly goals so that at any time, every person knew what they should be working on, and that the other group members were expecting it.

GitHub[1] was used, along with the Git source control system, to host all the project code and important files other than meeting notes. On several occasions this helped avert disaster when mistakes were made in developing code and writing the report, which are both collaborative tasks.

[1] http://github.com

Chapter 12

Conclusions

12.1 Achievements

As outlined in the Introduction, several goals were set at the start of this project, the main one being to implement cache modelling in the QEMU MIPS back-end, and then perform cache analysis and profiling using it.

12.1.1 MIPS Cache Support

The system provided emulates a MIPS cache, providing configuration options for L1 data and instruction caches, and an L2 integrated cache. Each cache is optional, and is completely configurable in terms of size, architecture, and replacement algorithm. The system is designed to be as fast as possible, without compromising code readability, and it was seen that it did not slow down QEMU to unusuable speeds.

12.1.2 Profiling and Analysis

A reliable logging format was implemented, and a set of Python utilities was developed to go with the system to facilitate analysis. These utilities include methods to sort log files and present data in several graphical ways. This allows different cache designs and configurations to be compared and their performance analysed. In addition, a live cache profiling tool was developed that displays the cache usage while QEMU is running, which greatly helps to visualise a program's cache impact.

12.1.3 Kernel Configureation and GNU Applications

Finally, a slimmed down Linux kernel was developed with the intention of being used as a base on which to run applications under QEMU. This greatly speeds up the process, and makes profiling easier.

12.2 Future Work

12.2.1 Improve TLB Integration

The current system has vitually no integration with the MIPS TLB, and only uses it once to translate between virtual and physical addresses. This could be changed in future, and a more efficient way of getting physical addresses would be beneficial. There is still a problem with some addresses being unable to be translated, and it is unclear whether this is a problem in this project's code, or in the MIPS TLB code.

This could also potentially include some form of process monitoring by way of the ASID, although it is still unclear how useful this would be, or if it would be possible. However, it is another area to investigate further.

12.2.2 MIPS64

Perhaps one useful extension to the current system would be implementing support for MIPS64, as currently only 32 bit MIPS is supported. Essentially, this would require updating the storage structures to allow for the 64 bit address space, and modifying the translation code to also call helpers for 64 bit instructions.

12.2.3 QEMU Monitor Additions

One discussion during this project was whether to implement a QEMU Monitor hook into the cache. Monitor is a part of QEMU that allows the running system to be inspected in real time, and thus it could be useful for monitoring the cache. However, due to time constraints, other goals were prioritised over this, and it remains something to investigate.

12.2.4 Live Display Enhancements

Finally, the live display could be enhanced in several ways. For example, methods could be added to clear the display (re-set counters to 0) while the program runs. This could allow certain parts of program execution to be specifically targetted for analysis.

12.3 Final Conclusions

The Gantt chart drawn up at the start of the project was revised several times as we found ourselves ahead of our expectations. All of the main targets were met in time, and other goals that were not initially set were also accomplished, such as live data analysis.

The project has met the specifications set by the customer, and has been developed so that it will be easily integrated into a real world work flow. A system for emulating the MIPS CPU cache has been created, and a set of Python tools come with it to make analysis of the data much easier. Future work involves improving the address translation routines, the live data analysis can be extended in several ways, and the system could be extended to work with MIPS64 as well.

Appendix A

Project Brief

School of Electronics And Computer Science
ELEC6050 MEng Group Design Project

Project Specification And Plan

mproving CPU performance modelling in QEMU

sor: Denis A Nicole, Steve R Gunn

lembers:
ahmoodi (dm4g10)
da Silva (rmds1g10)
Gupta (ng2g10)
epuch (rs5g10)
andon (at3g10)

er: Imagination Technologies Ltd.

Specification:

is a piece of small fast memory that is used by the CPU to store copies of recently used data items from mory. When looking for a data item the CPU will look in the cache first. If it finds the item here it has cache, if not it has "missed" the cache and it will get the item from main memory and the cache is Missing the cache reduces system performance and is something system designers try to avoid. Cache is used to see how many times the cache is hit or missed. It is an important tool when analysing the ance of systems, particularly embedded systems, and can be identify areas of code that need to be to make more efficient usage of the cache.

s an emulator and can be used to emulate an entire system for example GNU/Linux without the need for rdware. This makes it a popular choice amongst MIPS engineers and developers as it allows for rapid rototyping and debugging. Currently QEMU does not have any cache modelling support, which means le to be used to do cache profiling.

rall aims for the project therefore are to add MIPS cache support to QEMU, and then to perform cache on the Linux kernel and assorted GNU applications. This breaks the project into five main areas:

ache support to MIPS backend for QEMU.
The code needs to model L1 and L2 caches. L1 has separate data and instruction caches.
The code also needs to be able to model different cache types – direct, two-way/four-way set associate; along with different cache line sizes.
QEMU is designed as a fast emulator, so adding in any cache modelling support will slow it down. Therefore care needs to be taken when implementing the cache modelling code so that it is as fast as possible.

rch into useful MIPS Linux kernel configurations, and GNU applications that can be used for cache .

each of the MIPS Linux kernels, and associated file systems containing the GNU applications.

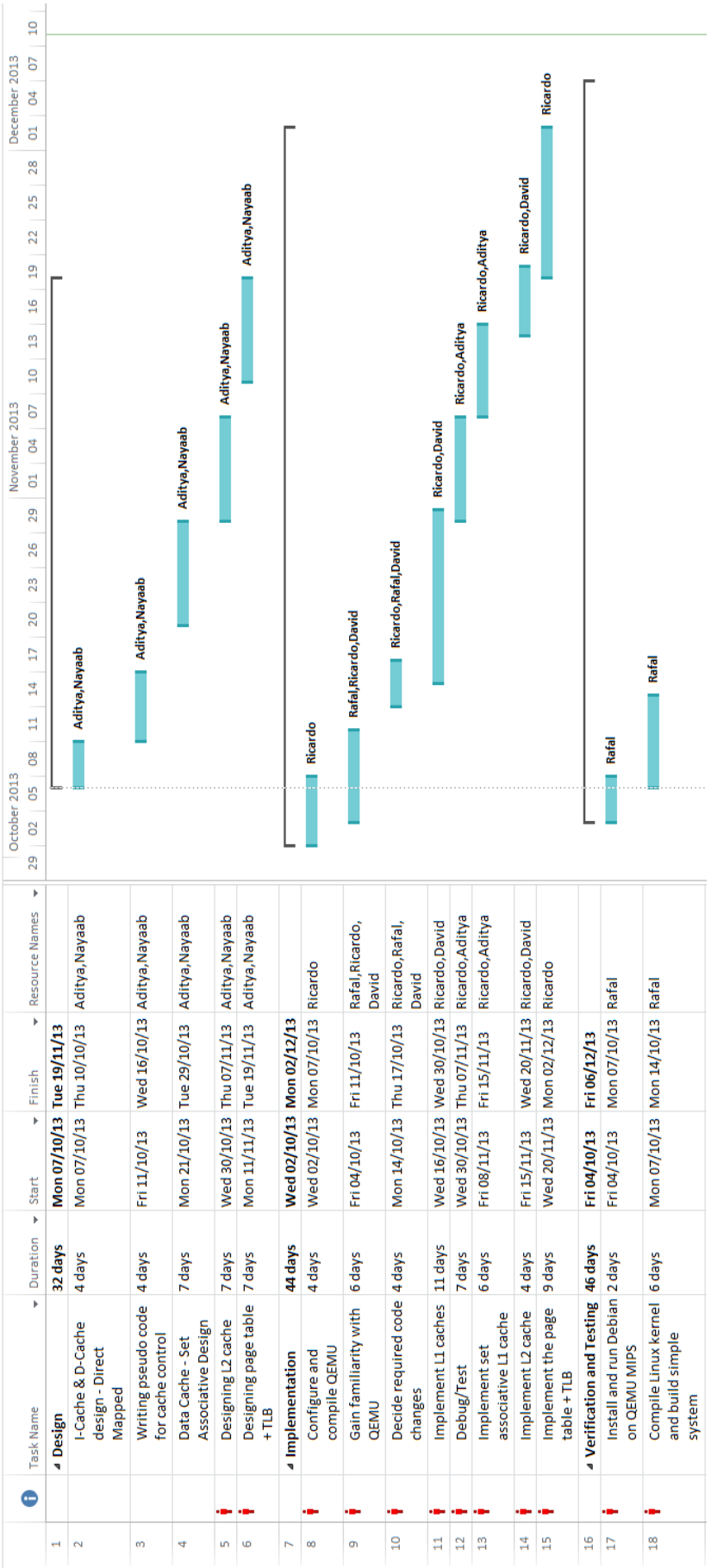
e kernels, and GNU applications in QEMU and get back cache profiling information.
This will look at different cache configurations, for example different cache line sizes, or set-associative sizes.

se the cache profile information, and produce some conclusions about how cache performance could be d.
The analysis will look at cache misses per instruction, cache refill penalty, memory write back rate or finding "hot" memory areas.
The conclusions will look at how the cache design could be improved, or how the code for the kernel or applications could be improved.

Appendix B

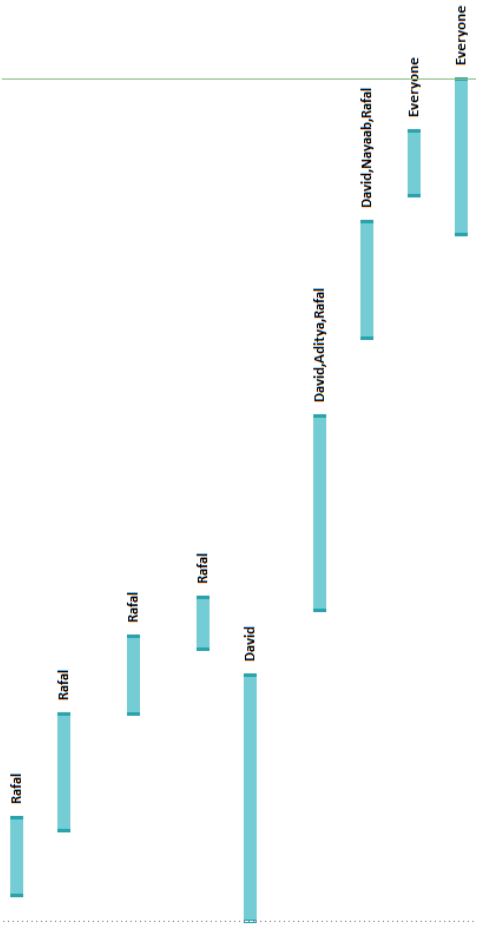
Gantt Chart

Please find the Gantt chart on the next 2 pages.



	Task Name	Duration	Start	Finish	Resource Names
1	**Design**	**32 days**	**Mon 07/10/13**	**Tue 19/11/13**	
2	I-Cache & D-Cache design - Direct Mapped	4 days	Mon 07/10/13	Thu 10/10/13	Aditya,Nayaab
3	Writing pseudo code for cache control	4 days	Fri 11/10/13	Wed 16/10/13	Aditya,Nayaab
4	Data Cache - Set Associative Design	7 days	Mon 21/10/13	Tue 29/10/13	Aditya,Nayaab
5	Designing L2 cache	7 days	Wed 30/10/13	Thu 07/11/13	Aditya,Nayaab
6	Designing page table + TLB	7 days	Mon 11/11/13	Tue 19/11/13	Aditya,Nayaab
7	**Implementation**	**44 days**	**Wed 02/10/13**	**Mon 02/12/13**	
8	Configure and compile QEMU	4 days	Wed 02/10/13	Mon 07/10/13	Ricardo
9	Gain familiarity with QEMU	6 days	Fri 04/10/13	Fri 11/10/13	Rafal,Ricardo, David
10	Decide required code changes	4 days	Mon 14/10/13	Thu 17/10/13	Ricardo,Rafal, David
11	Implement L1 caches	11 days	Wed 16/10/13	Wed 30/10/13	Ricardo,David
12	Debug/Test	7 days	Wed 30/10/13	Thu 07/11/13	Ricardo,Aditya
13	Implement set associative L1 cache	6 days	Fri 08/11/13	Fri 15/11/13	Ricardo,Aditya
14	Implement L2 cache	4 days	Fri 15/11/13	Wed 20/11/13	Ricardo,David
15	Implement the page table + TLB	9 days	Wed 20/11/13	Mon 02/12/13	Ricardo
16	**Verification and Testing**	**46 days**	**Fri 04/10/13**	**Fri 06/12/13**	
17	Install and run Debian on QEMU MIPS	2 days	Fri 04/10/13	Mon 07/10/13	Rafal
18	Compile Linux kernel and build simple system	6 days	Mon 07/10/13	Mon 14/10/13	Rafal

Figure B.1: Gantt chart, part 1



19	Run the system on QEMU MIPS	4 days	Wed 09/10/13	Mon 14/10/13	Rafal
20	Prepare simple tests for debugging the design	7 days	Mon 14/10/13	Tue 22/10/13	Rafal
21	Familarize with existing regression tests	4 days	Wed 23/10/13	Mon 28/10/13	Rafal
22	Write more thorough debugging tests	4 days	Mon 28/10/13	Thu 31/10/13	Rafal
23	Investigate Valgrind and other testing software	15 days	Mon 07/10/13	Fri 25/10/13	David
24	Cache Profiling L1 Caches	11 days	Thu 31/10/13	Thu 14/11/13	David,Aditya,Rafal
25	Cache Profiling L2 Cache	7 days	Thu 21/11/13	Fri 29/11/13	David,Nayaab, Rafal
26	Performance Testing entire system	5 days	Mon 02/12/13	Fri 06/12/13	Everyone
27	**Report Writing**	**8 days**	**Fri 29/11/13**	**Tue 10/12/13**	**Everyone**

Figure B.2: Gantt chart, part 2

Appendix C

File Listing

Below is a full tree of the files included in the archive submitted along with this report.

In the QEMU directory, only the files modified or added in this project are listed here – the ones that were changed are highlighted in orange.

GDPPython/

- CachePloting2DBar.py
- CachePloting2DLine.py
- CachePloting3D.py
- Cache_Profiling_Performance_Analysis.xlsx
- Log_File_Sorter.py
- QEMU-off-line-Profiling.py
- QEMU-off-line-Analysis.py

qemu/

- vl.c
- qemu-options.hx
- linux-user/
  - main.c
- target-mips/
  - Makefile.objs

  - cache.c
  - cache.h
  - cache_helper.c
  - cpu.h
  - gnuplot_i.c
  - gnuplot_i.h
  - helper.c
  - helper.h
  - mips-cache-opts.c
  - mips-cache-opts.h
  - replacement_lfu.c
  - replacement_lru.c
  - replacement_rnd.c
  - testbench/Makefile
  - testbench/lru_access.c
  - testbench/runTests.c
  - testbench/runTests.h
  - translate.c
  - translate_init.c

eric_linux/

- README
- automated_test.sh
- datafs.ext2
- rootfs.ext2
- start_linux
- vmlinux

Appendix D

Project Meetings and Planing

Followings show some examples of the notes that have been taken at the end of each meeting and some of the planing that was made.

Meeting1: 04/10

By Ric da Silva on Friday, 4 October 2013 at 12:59

First had skype call with Denis:

- prioritise cache

Call with Andrew:

- how to divide task (1 person cache design, 2 people QEMU, 2 people building kernels & analysing them).
- start with prebuilt kernel, but will need to optimise it for our design
- look at buildroot
- use lots of graphs for performance analysis

Weekend tasks for Tuesday

- Ric, Aditya: QEMU readup - overall structure, how to create modules, how it treats things like caches
- Aditya, Nayaab: How caches on MIPS work, read ImgTec specification
- Aditya, Nayaab: L1 cache design for the project - spec for what do we need to implement first
- Rafal: MIPS kernel research - what kernels can we use, how to build / compile them, how can they be slimmed down
- David: Investigate performance analysis - what programs can we use for cache profiling

Do this: **PREPARE LIST OF QUESTIONS FOR ANDREW**
Next meeting: Monday 7th
- Finalise Gantt chart
- Finish Specification
- Discuss weekend progress and Andrew question

Figure D.1: Meeting minutes of 04/11/2013

Meeting2: 08/10/2013

By Ric da Silva on Tuesday, 8 October 2013 at 14:33

Met Andrew:
Asked lots of questions,
Learnt about how QEMU works (target instr -> decode (via binutils) -> micro-operations -> translation blocks -> executed on host)

- The project is software base, hardware architecture knowledge is required
- Will need to modify the code in target-mips folder
- I-cache will be 'virtual' (just a look up table), D-cache will be real.
- Implement our cache stuff as a TCG helper (eg helpers.c)
- See the TLB helper for how to do a helper, make cache similar
- ASIDs can be used to identify processes (for analysis purposes)
- Use different kernel config options
- **Can't** use valgrind etc for analysis
- -> Dump hits/misses into a file and analyse later
- See page 71 of SMR for cache config fields
- See page 94 - cache prefix and sync instructions

First task - get something basic running
Later - implement cache control instructions
Big conclusion - we need a LOT more focus on QEMU
Progress since last week:

- Begun to understand QEMU backend system
- Understood the overall functionality of QEMU, ie, it does not model the hardware, but rather performs a sort of target-host instruction translation.
- Configured/compiled standard QEMU
- Built and run a 'stock' debian mips linux kernel
- Designed I- and D-caches as direct mapped caches, and wrote pseudo code. This was beneficial from a learning point of view
- Investigated MIPS cache functions
- Learnt about valgrind and cachegrind. Discovered that they won't really be useful for the project - now looking for different ways to analyse
- Discussed likely required code changes with Andrew from Imagination Tech
- Overall gained clarity on the project objectives. We were unsure about them before due to unfamiliarity with this area of work.

Goals:

- Everyone needs to become familiar with the MIPS target QEMU backend. Look in the target-mips folder.
- Investigate the QEMU TLB helper code
- Start setting up a custom Kernel

Plan:

- This Thursday - meet to join our knowledge about QEMU
- Have a semi-decent understanding of QEMU by Friday meeting

Figure D.2: Meeting minutes of 04/11/2013

Meeting 01/11/2013

By David Ma on Friday, 1 November 2013 at 17:03

Future work to be done in each area:

- **Design**:

1. work on pre fetching the data
2. 4 way set associative cache design
3. Random replacement cache policy design

- **Implementation**

1. dumping data for live performance analysis
2. L2-chache implementation
3. set associativity for both levels of cache
4. dumping log data for running test program only

- **Performance**

1. use excel for current off line analysis
2. work on live performance on python
3. use the benchmark programs for performance analysis and compare the results with current results available on the net

- **Extra possibilities**

1. saving the data on the cache
2. use multi-bank caches in the design

- work on pre fetching the data
- 4 way set associative cache design
- Random replacement cache policy design

Figure D.3: Meeting minutes of 01/11/2013

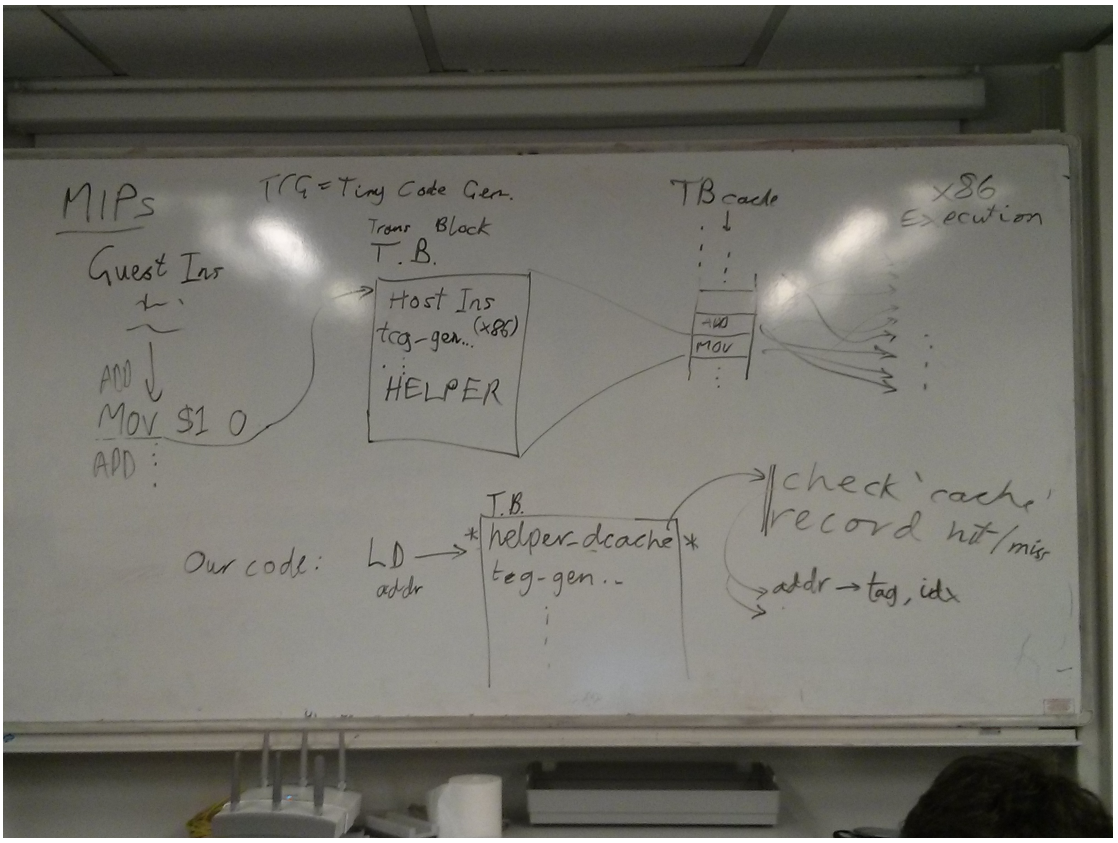


Figure D.4: Meeting on 08/11/2013

Appendix E

Cache Performance Figures

Following figures show the cache performance analysis of simulating chess game on QEMU.

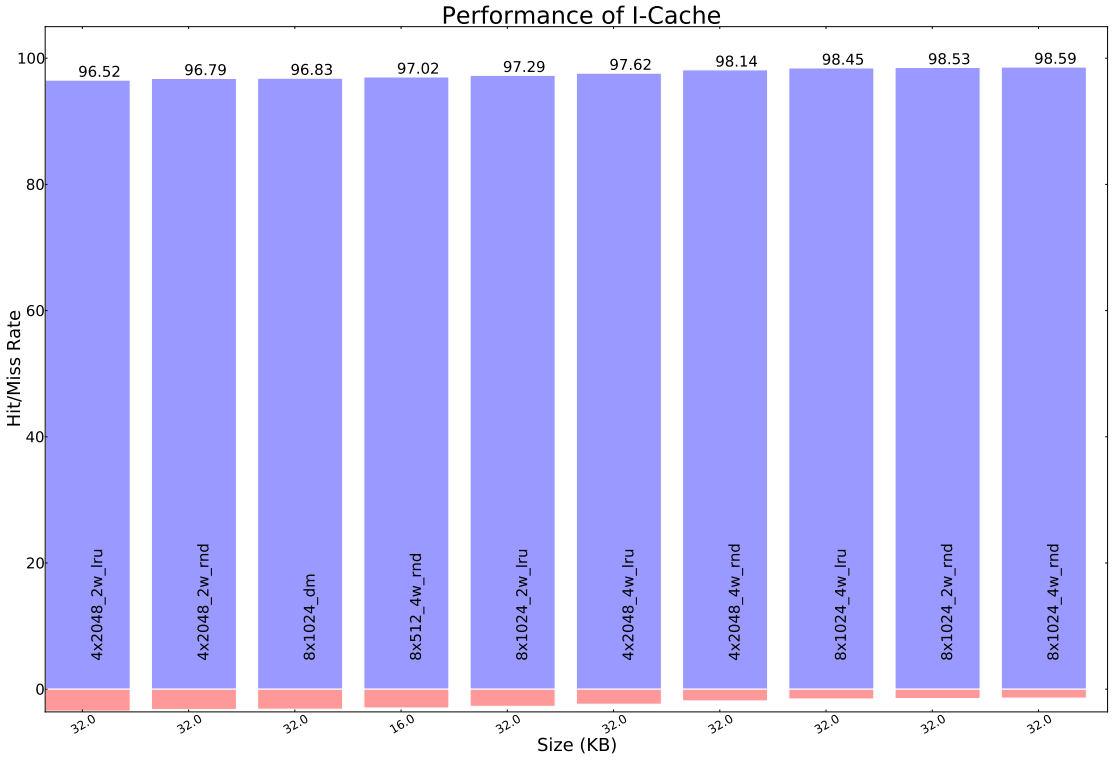


Figure E.1: Chess game best I-Cache designs

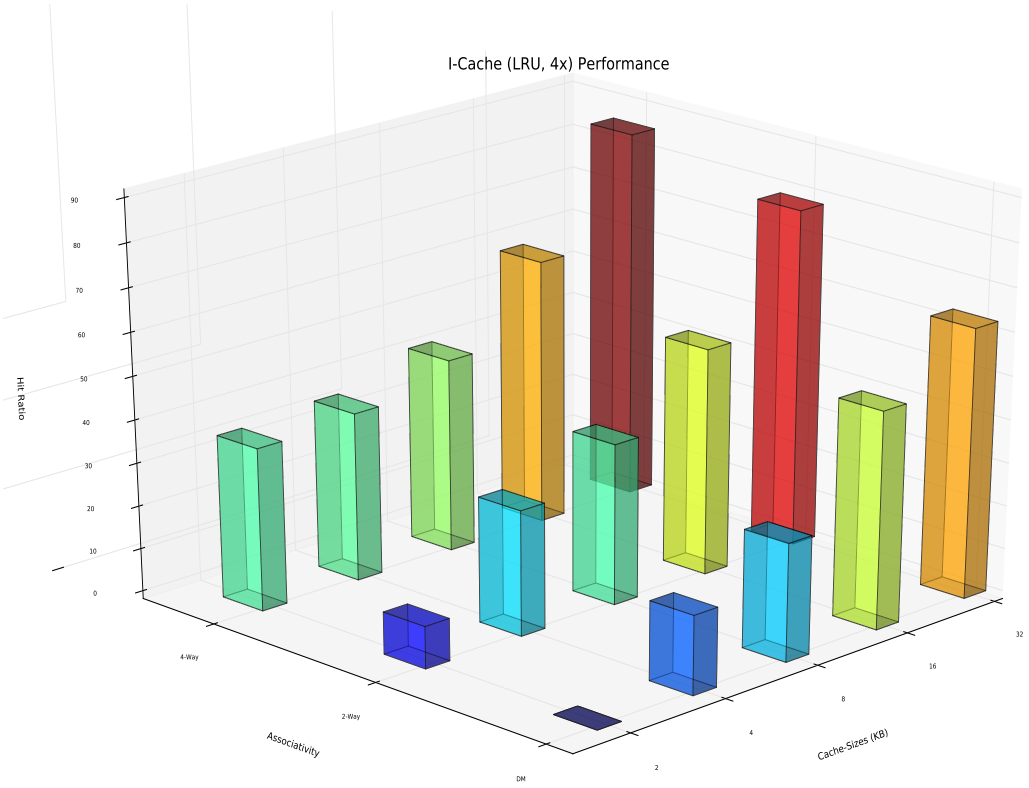


Figure E.2: Chess game I-Cache designs Associativity-Size 3D plot

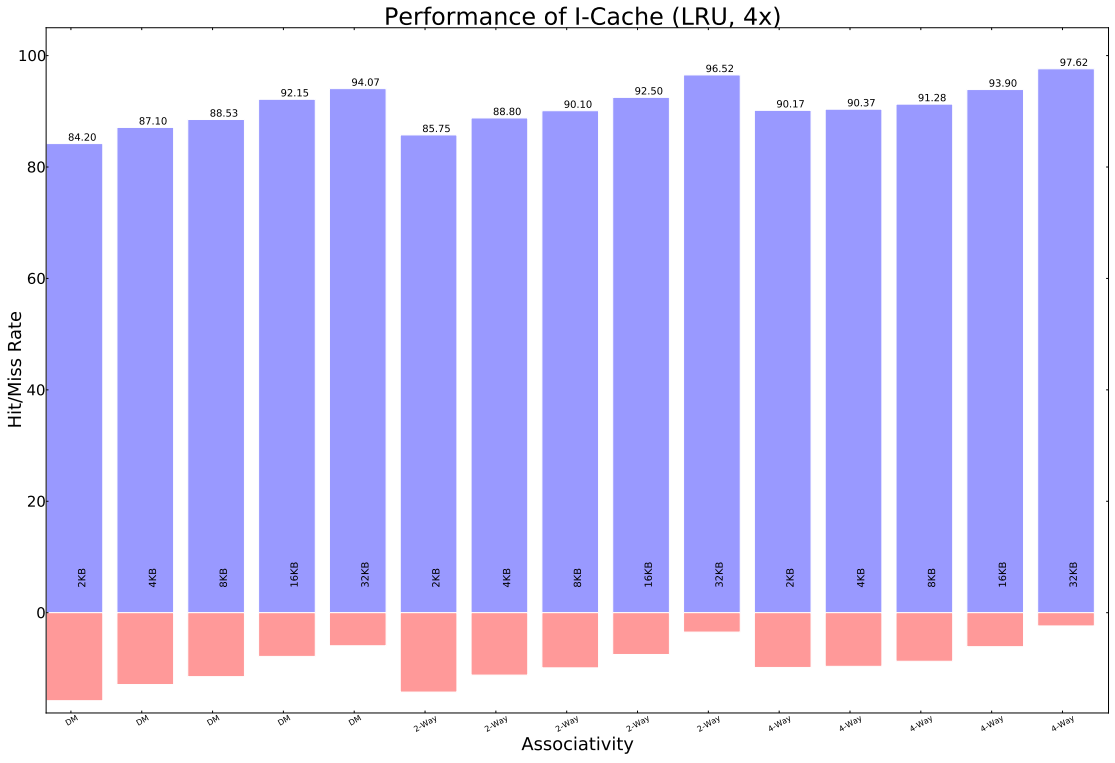


Figure E.3: Chess game I-Cache designs Associativity-Size Bar plot

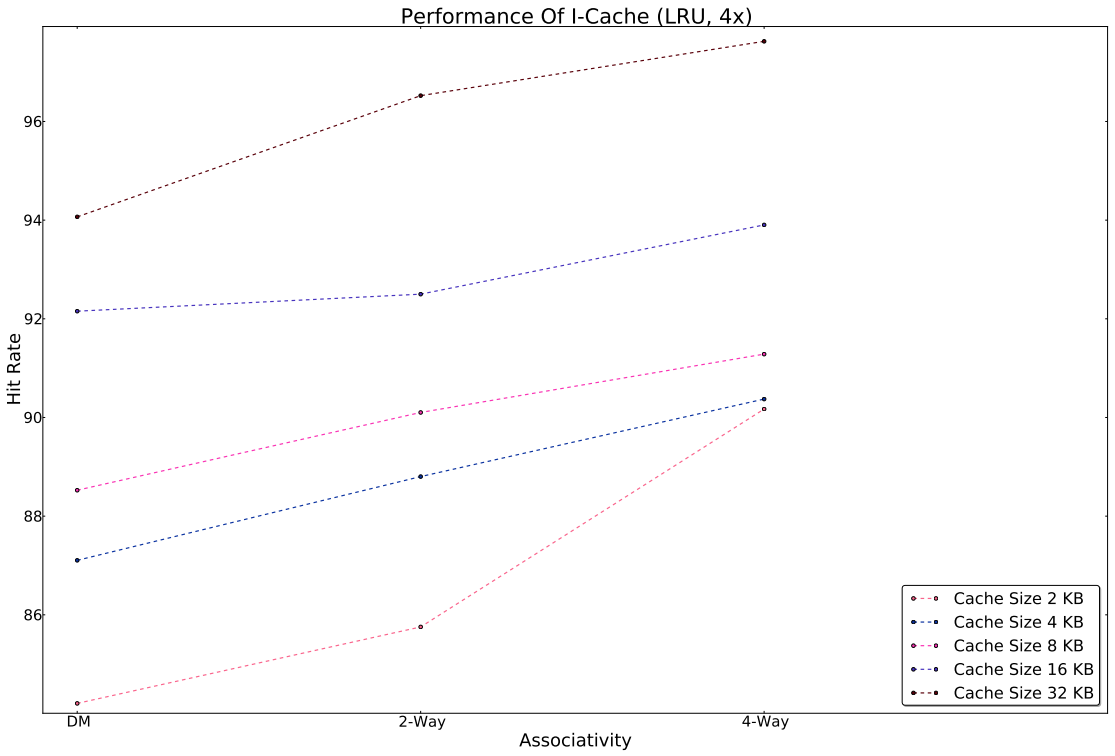


Figure E.4: Chess game I-Cache designs Associativity-Size Line plot

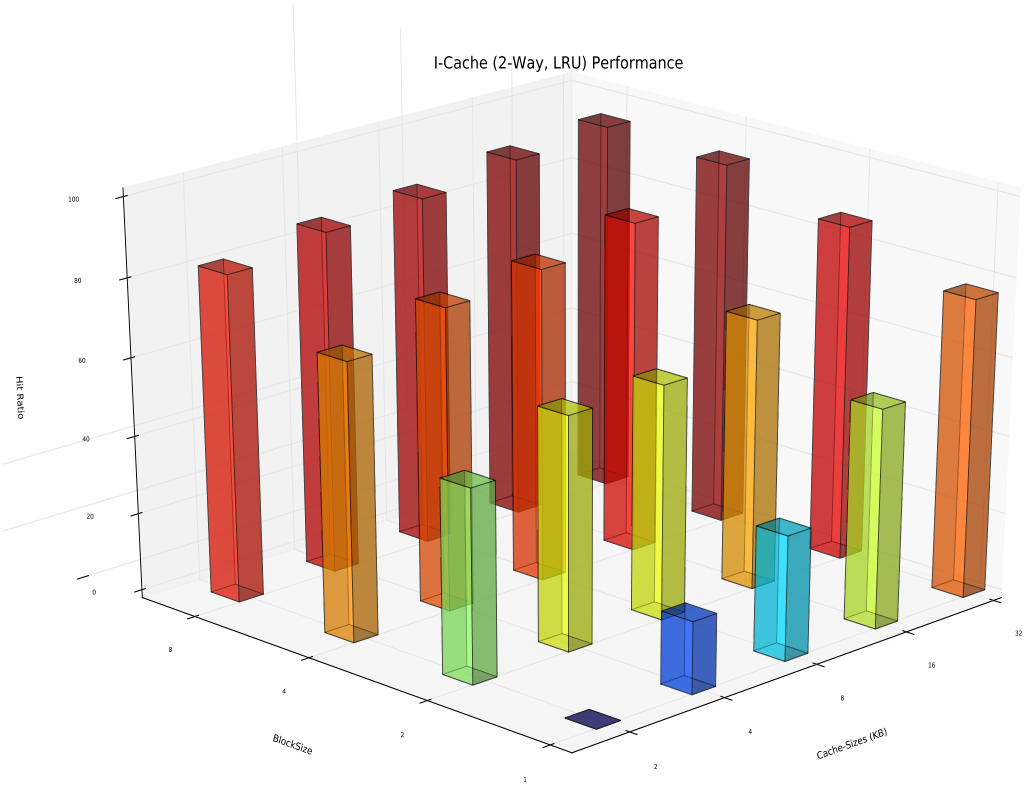


Figure E.5: Chess game I-Cache designs Block-Size-Size 3D plot

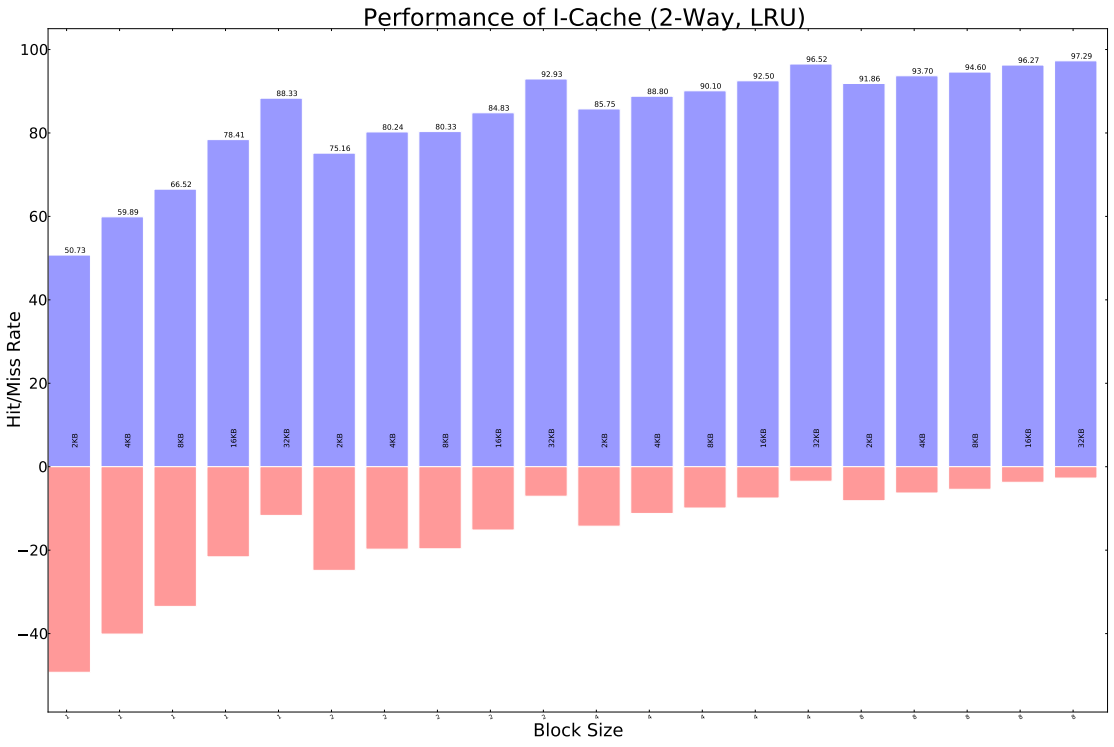


Figure E.6: Chess game I-Cache designs Block-Size-Size Bar plot

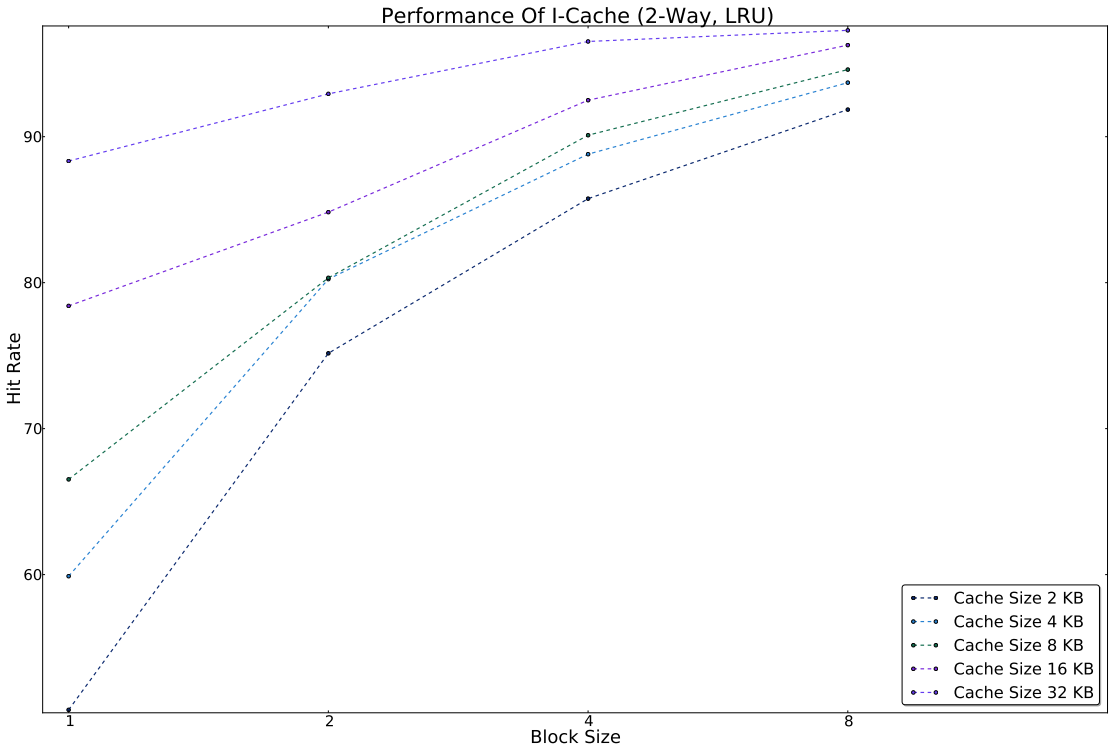


Figure E.7: Chess game I-Cache designs Block-Size-Size Line plot

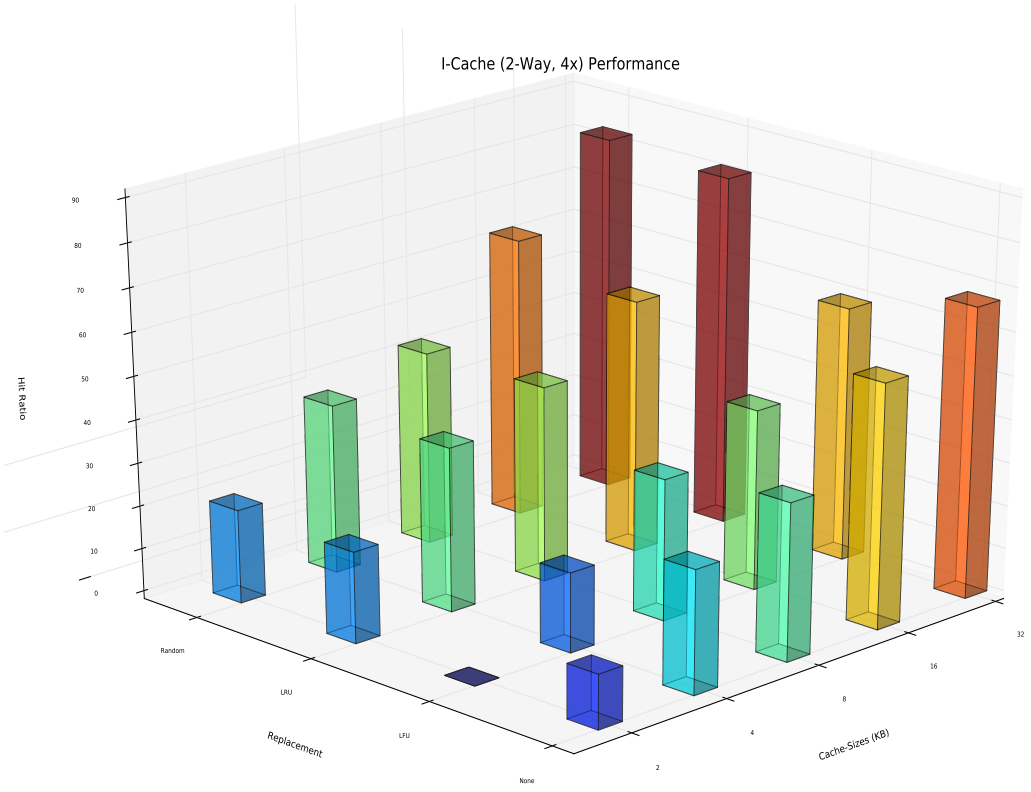


Figure E.8: Chess game I-Cache designs Replacement-Size 3D plot

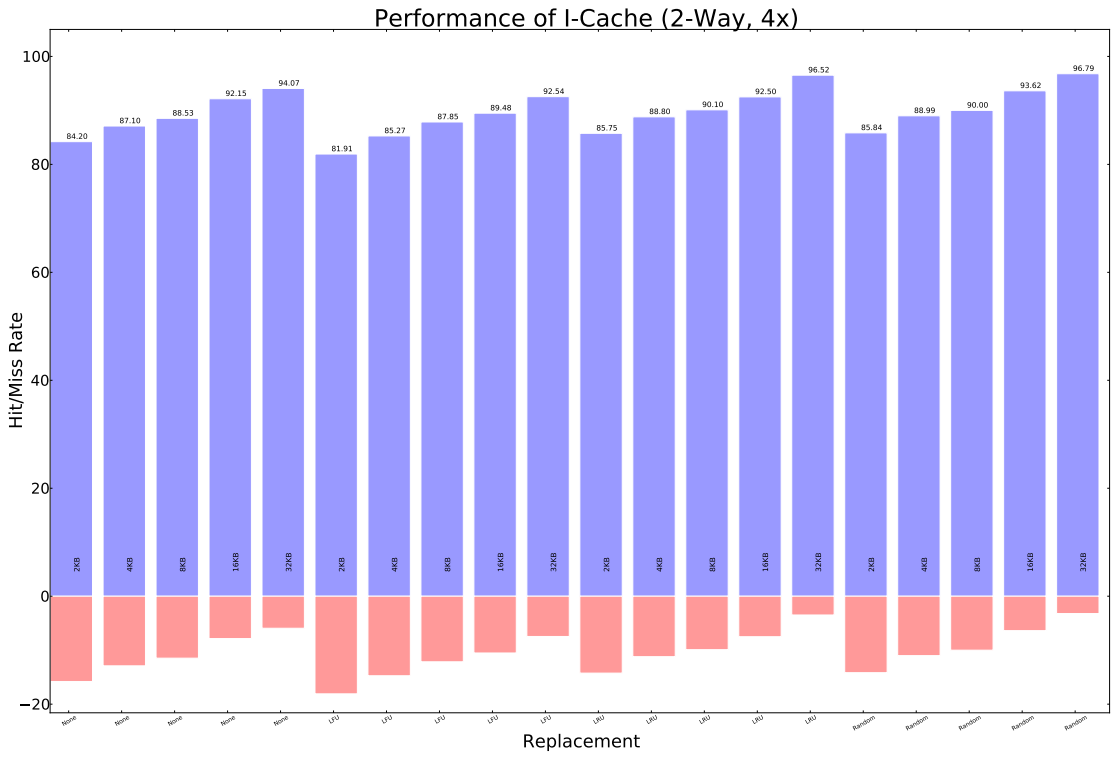


Figure E.9: Chess game I-Cache designs Replacement-Size Bar plot

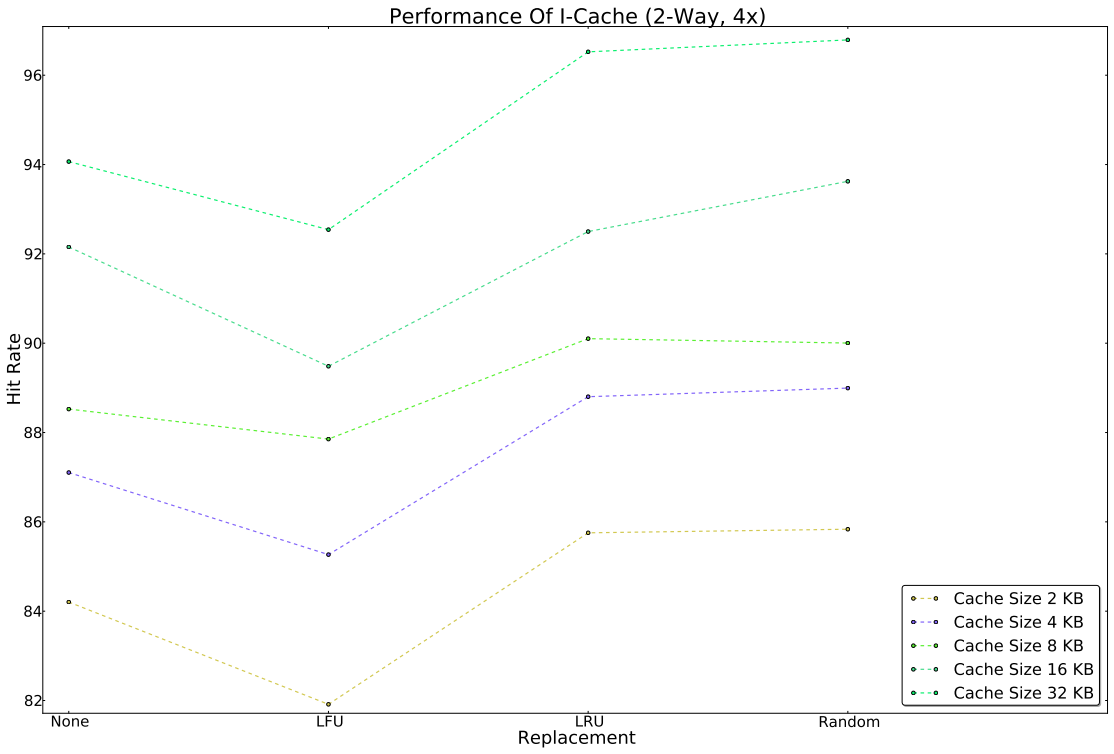


Figure E.10: Chess game I-Cache designs Replacement-Size Line plot

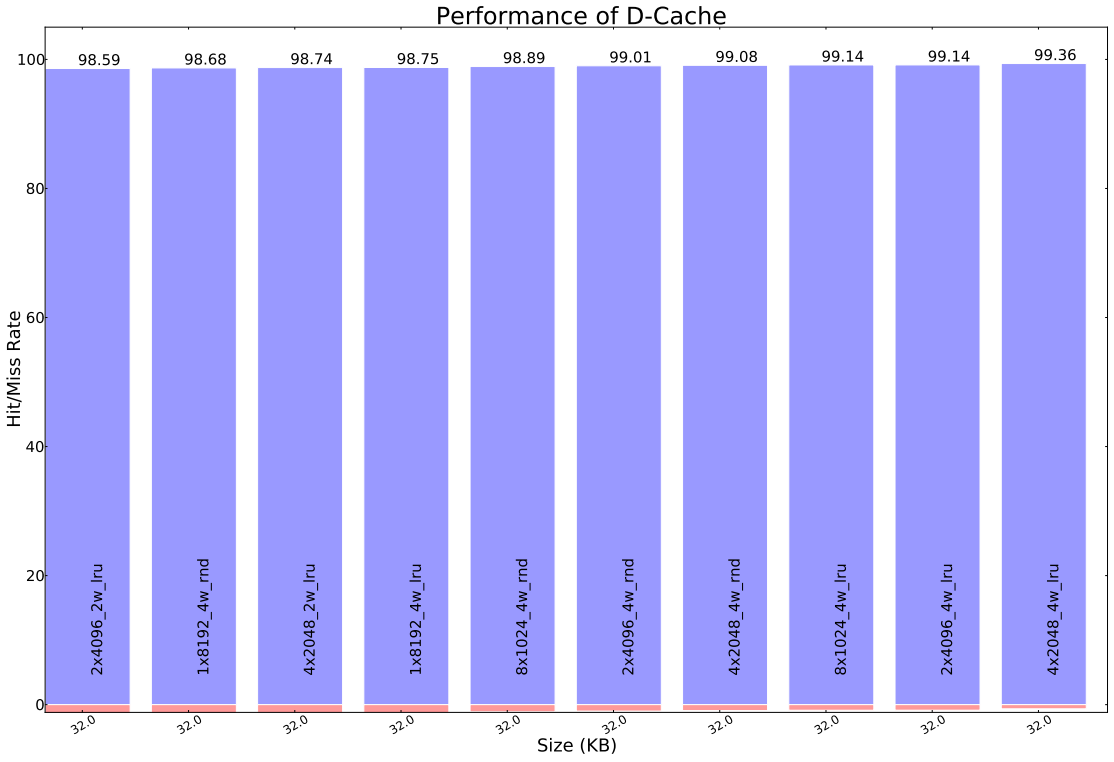


Figure E.11: Chess game best D-Cache designs

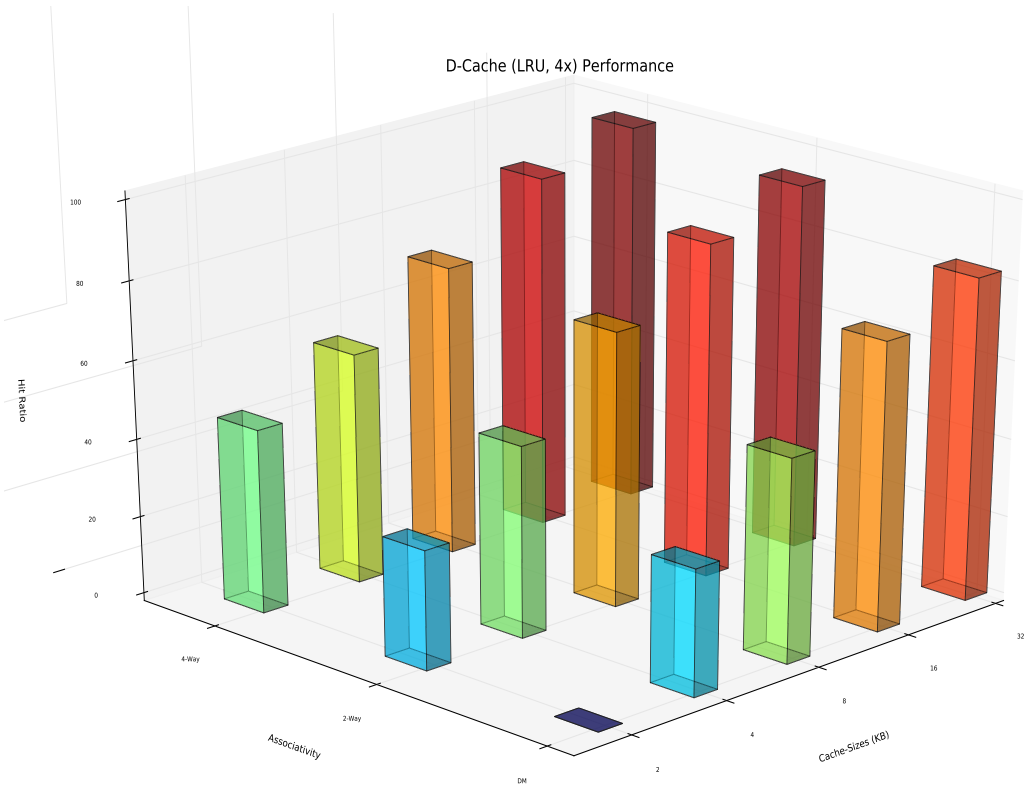


Figure E.12: Chess game D-Cache designs Associativity-Size 3D plot

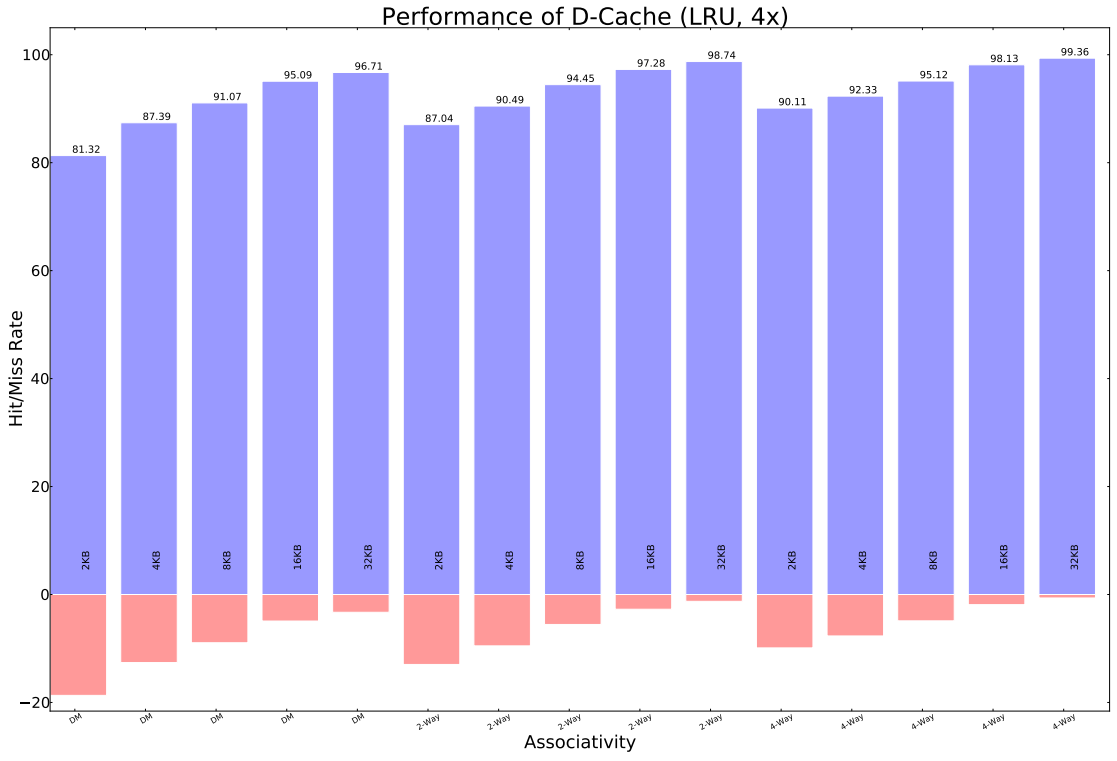


Figure E.13: Chess game D-Cache designs Associativity-Size Bar plot

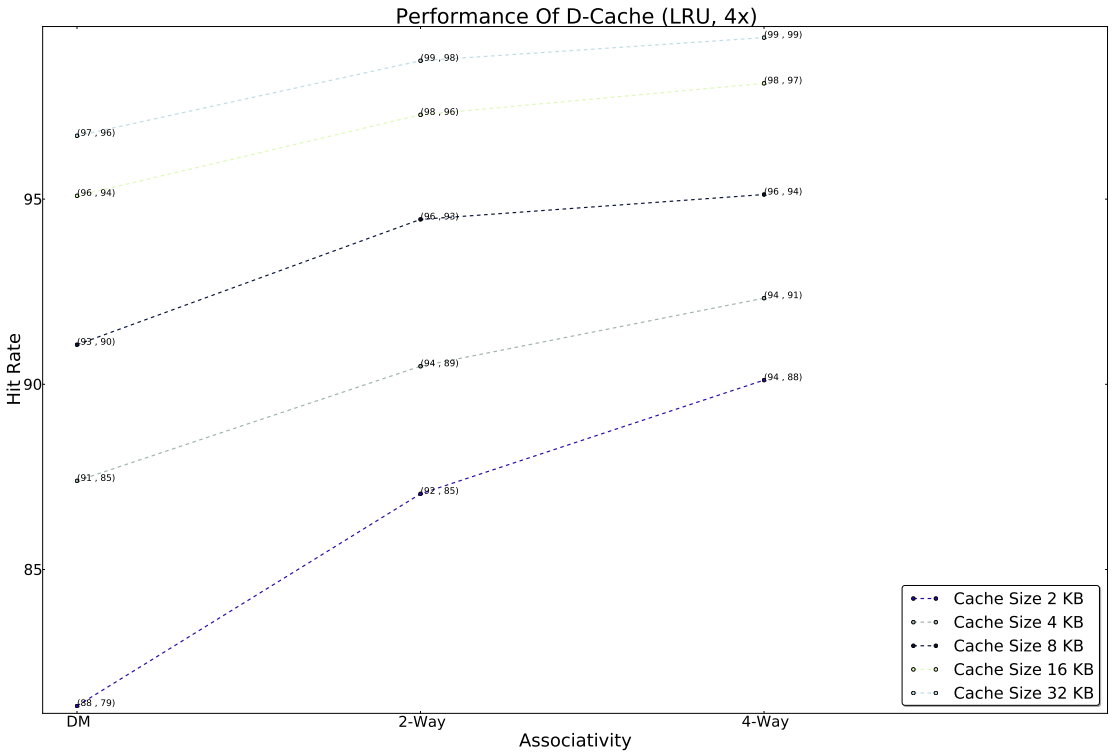


Figure E.14: Chess game D-Cache designs Associativity-Size Line plot

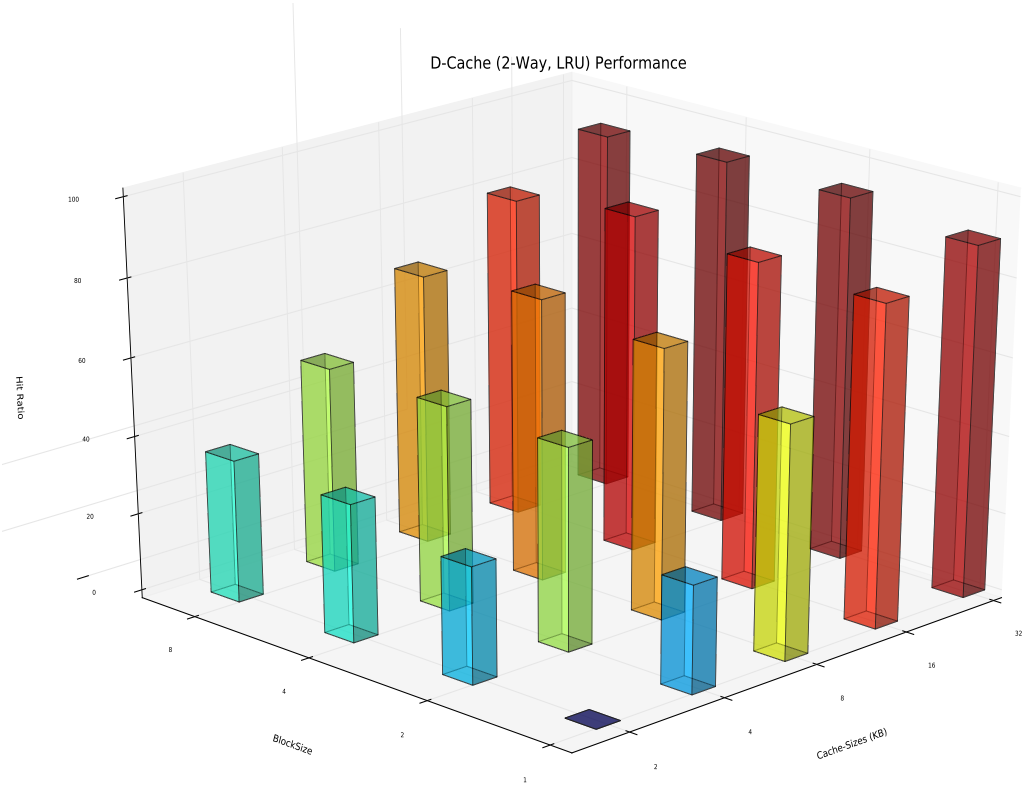


Figure E.15: Chess game D-Cache designs Block-Size-Size 3D plot

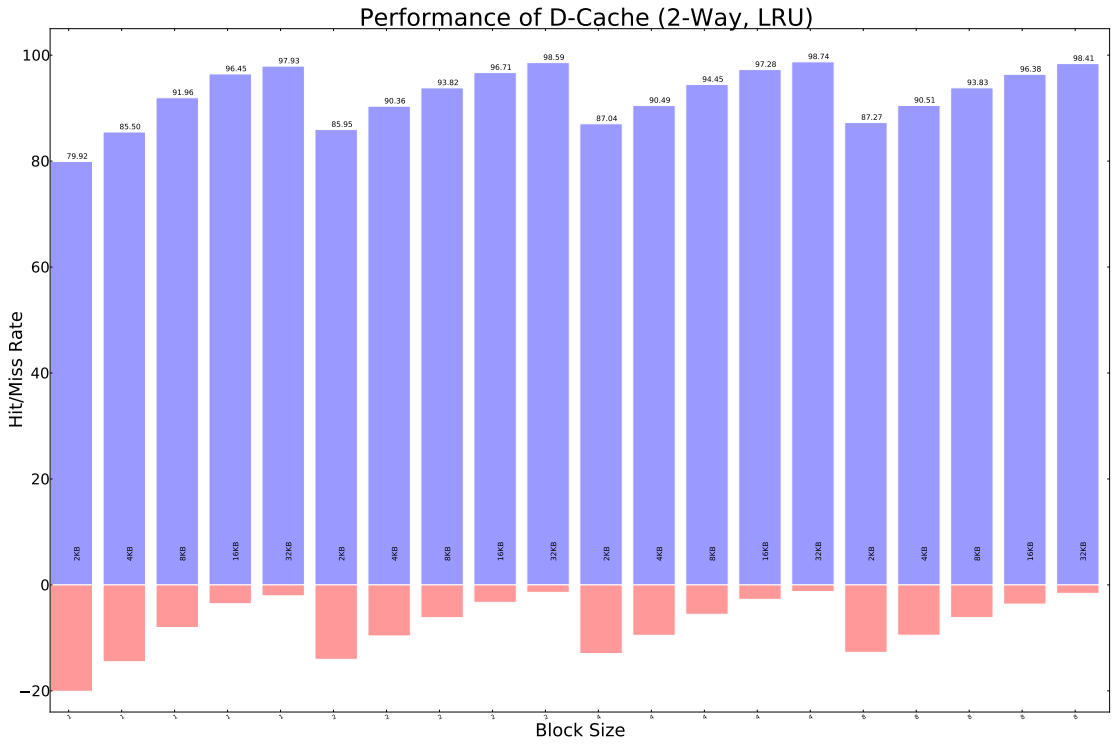


Figure E.16: Chess game D-Cache designs Block-Size-Size Bar plot

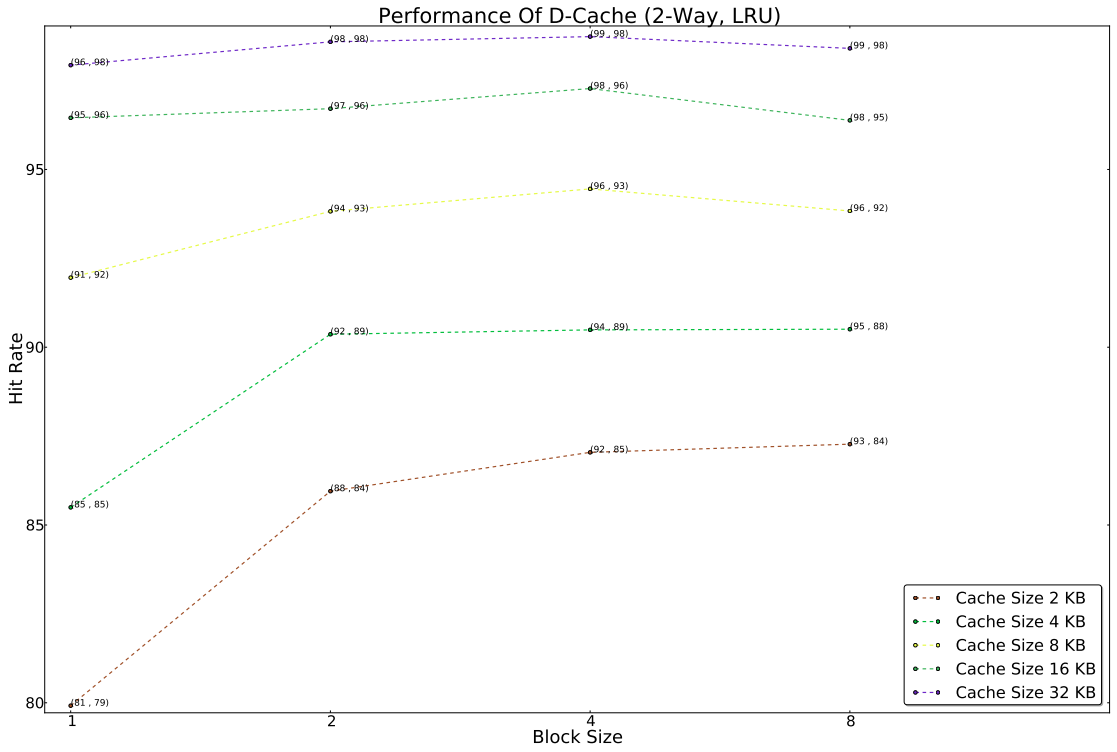


Figure E.17: Chess game D-Cache designs Block-Size-Size Line plot

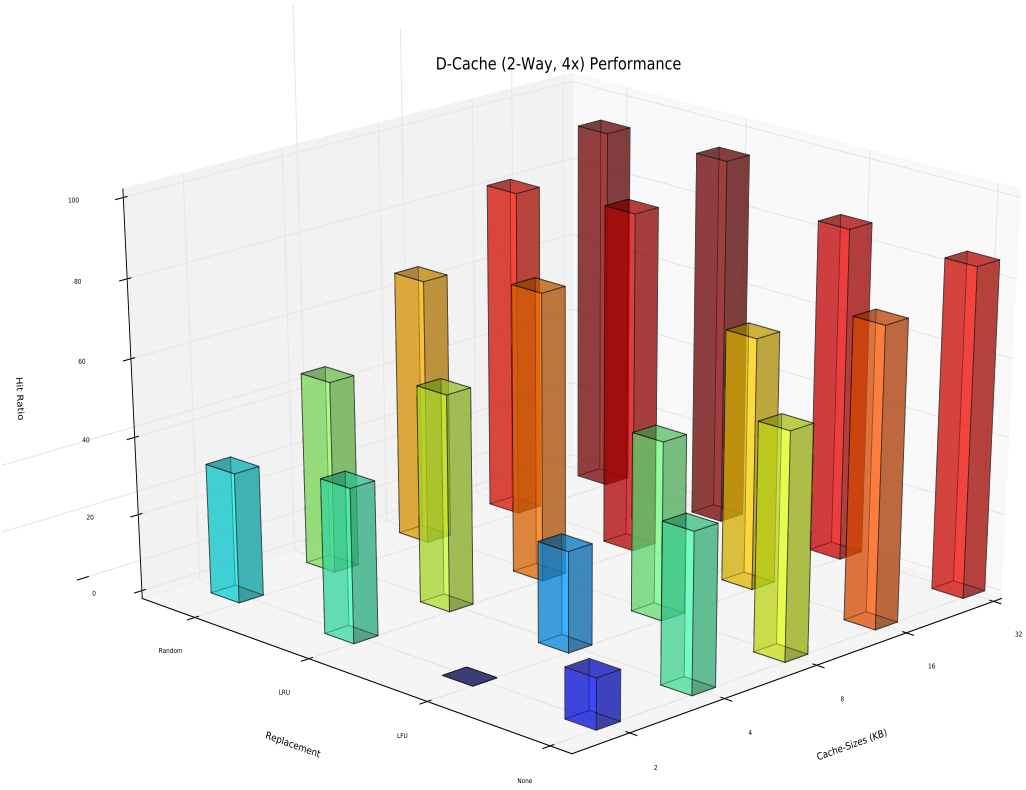


Figure E.18: Chess game D-Cache designs Replacement-Size 3D plot

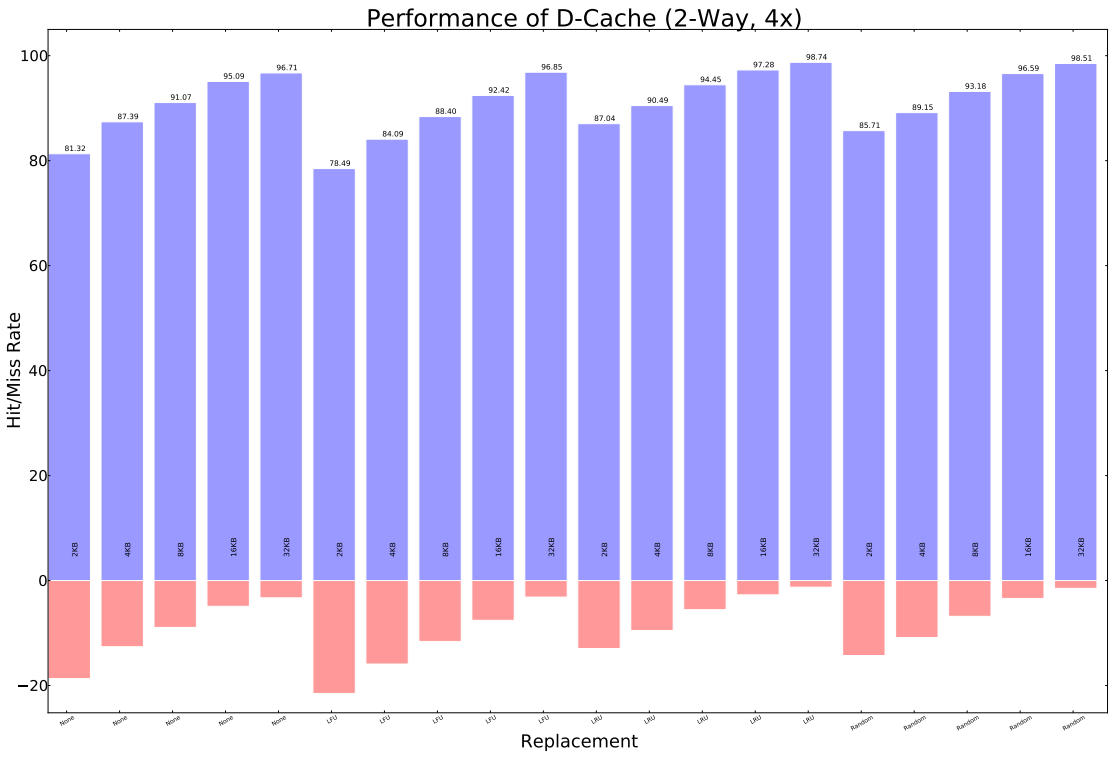


Figure E.19: Chess game D-Cache designs Replacement-Size Bar plot

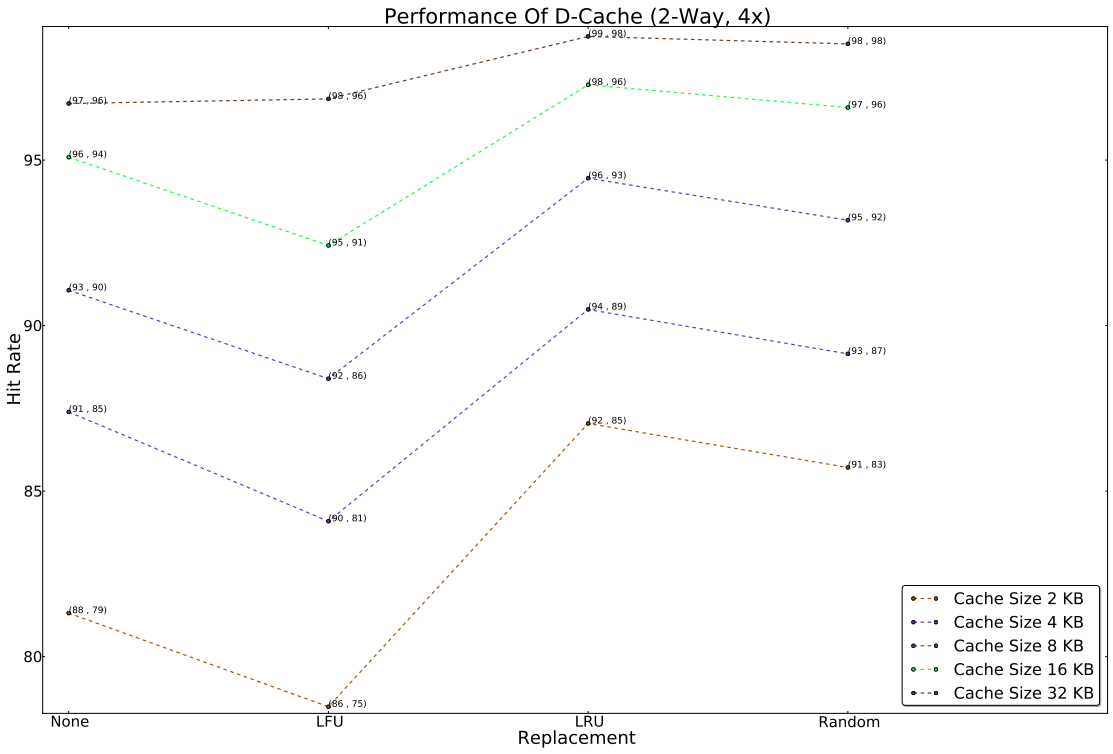


Figure E.20: Chess game D-Cache designs Replacement-Size Line plot

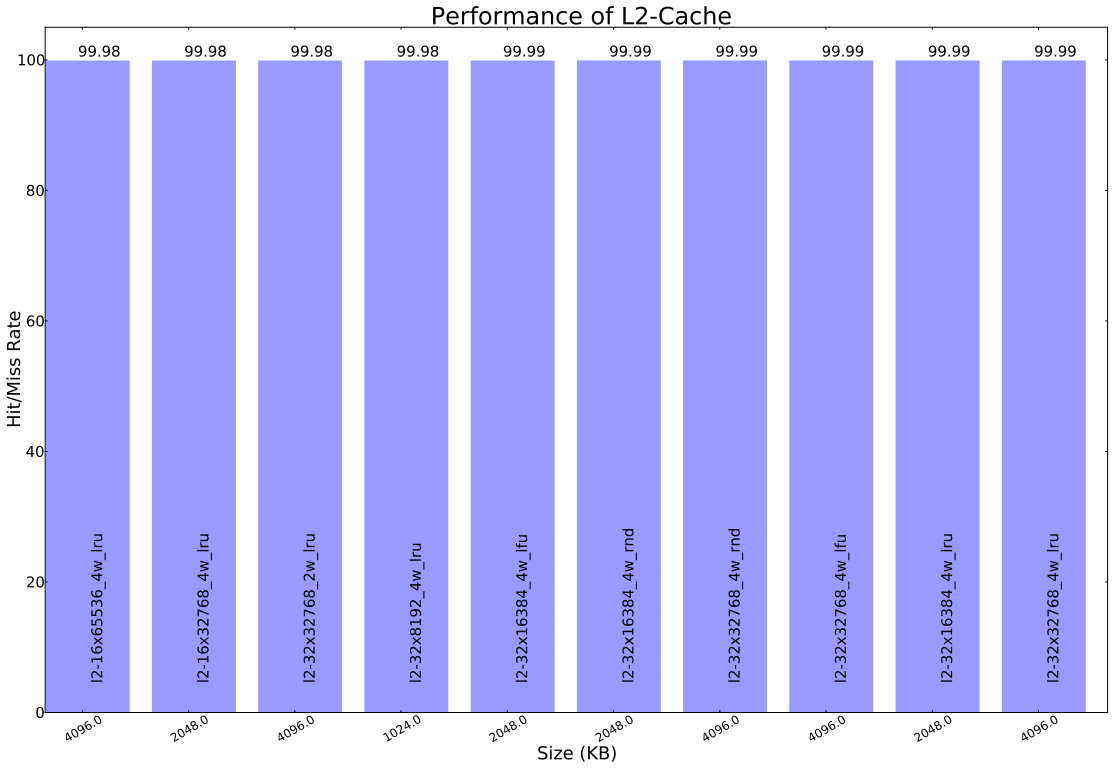


Figure E.21: Chess game best L2-Cache designs

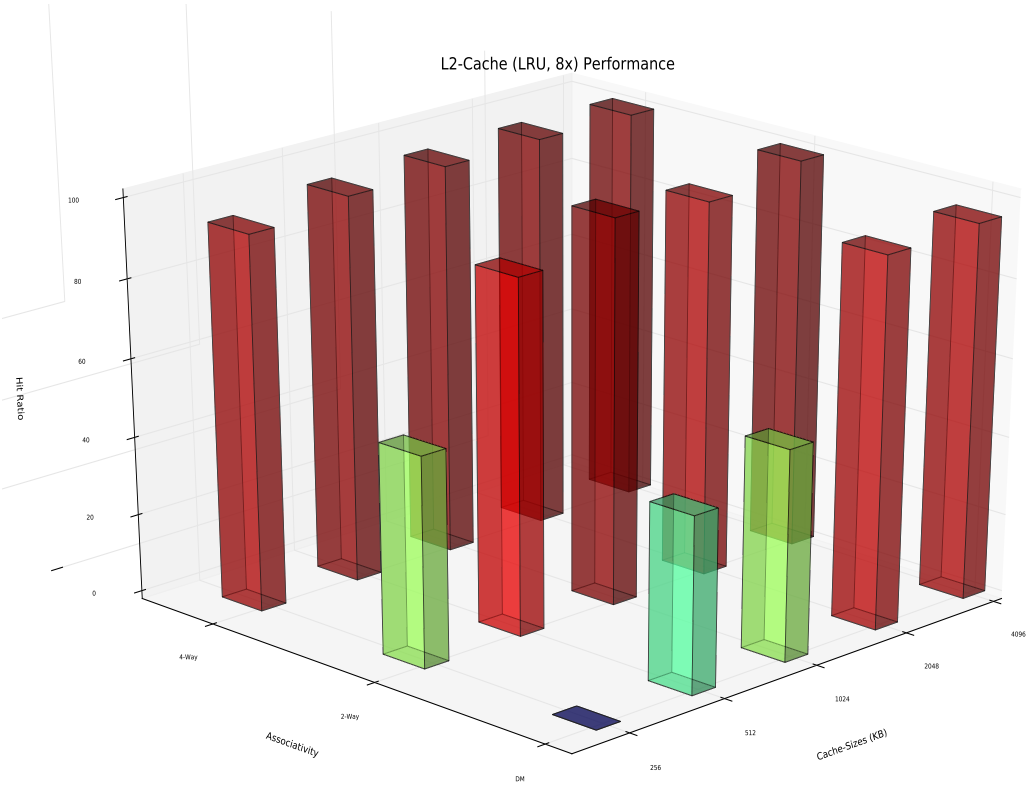


Figure E.22: Chess game L2-Cache designs Associativity-Size 3D plot

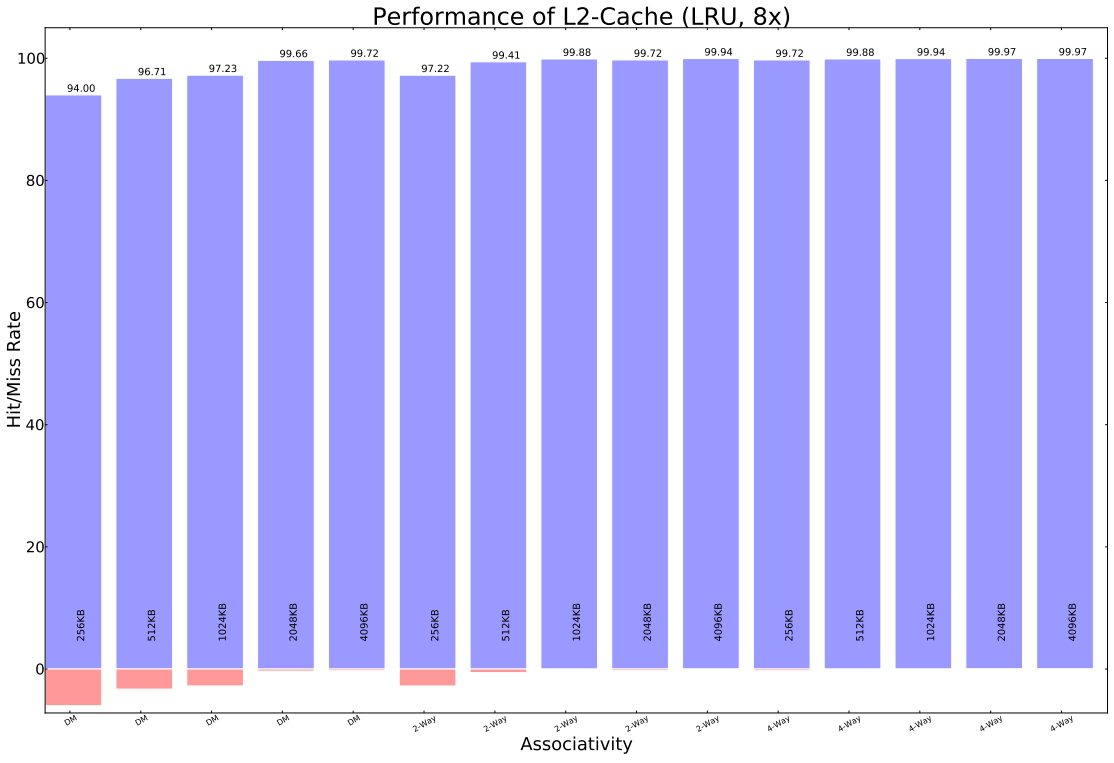


Figure E.23: Chess game L2-Cache designs Associativity-Size Bar plot

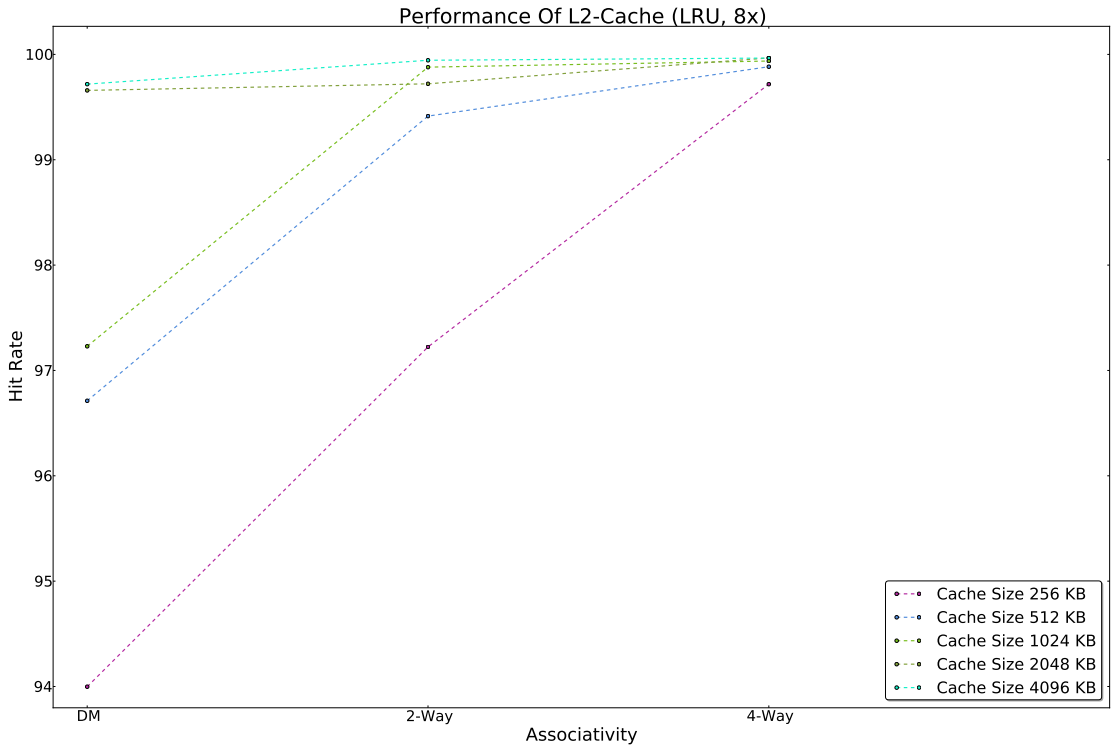


Figure E.24: Chess game L2-Cache designs Associativity-Size Line plot

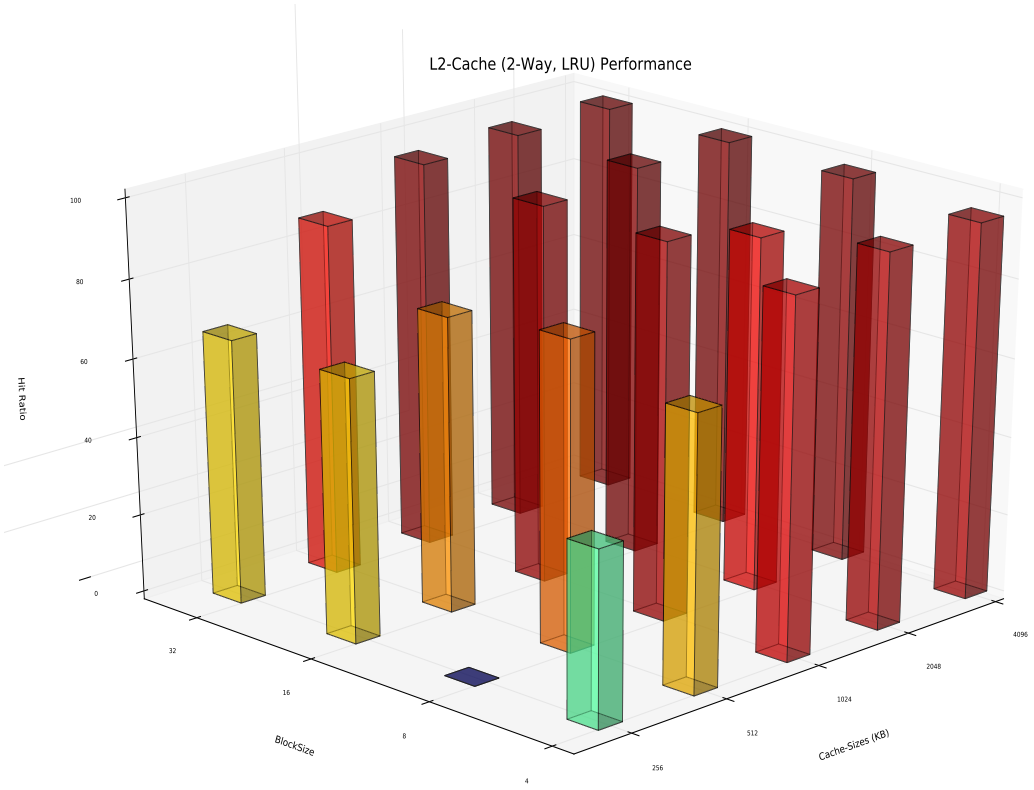


Figure E.25: Chess game L2-Cache designs Block-Size-Size 3D plot

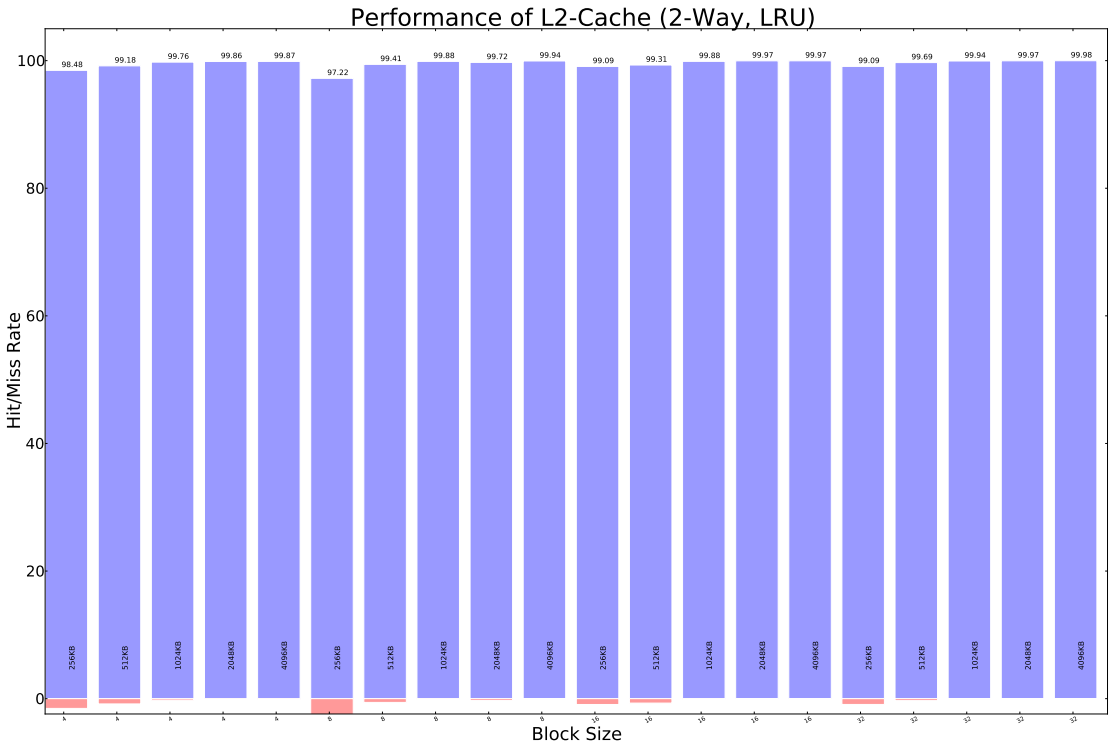


Figure E.26: Chess game L2-Cache designs Block-Size-Size Bar plot

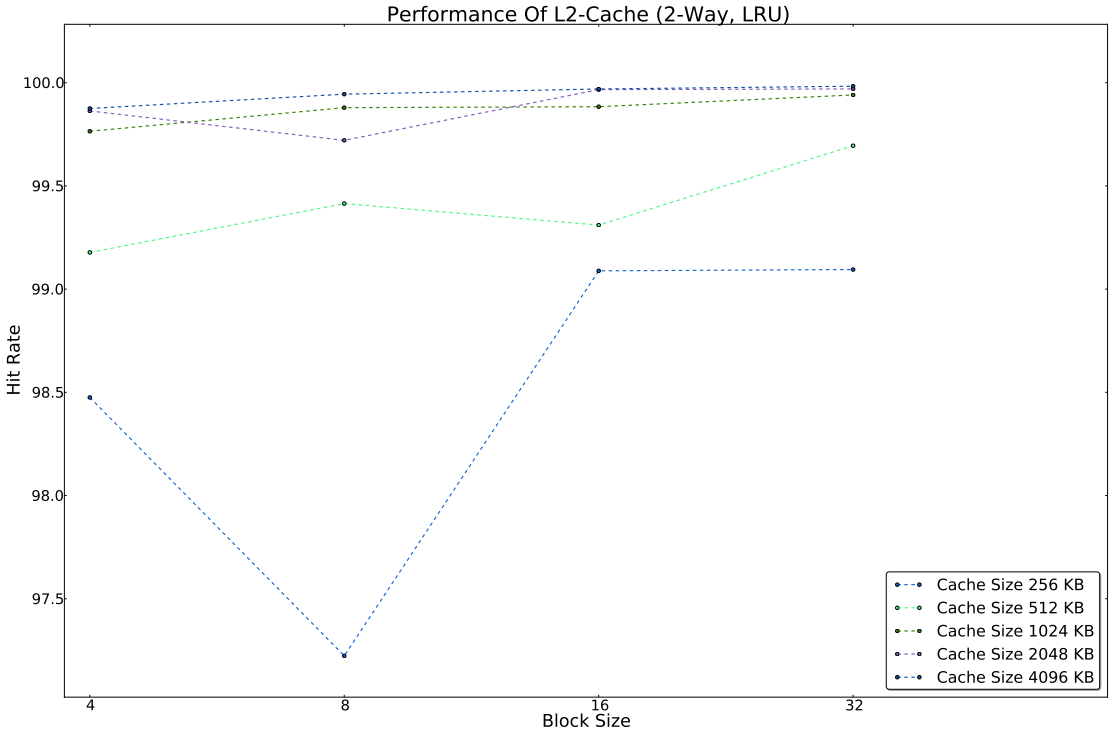


Figure E.27: Chess game L2-Cache designs Block-Size-Size Line plot

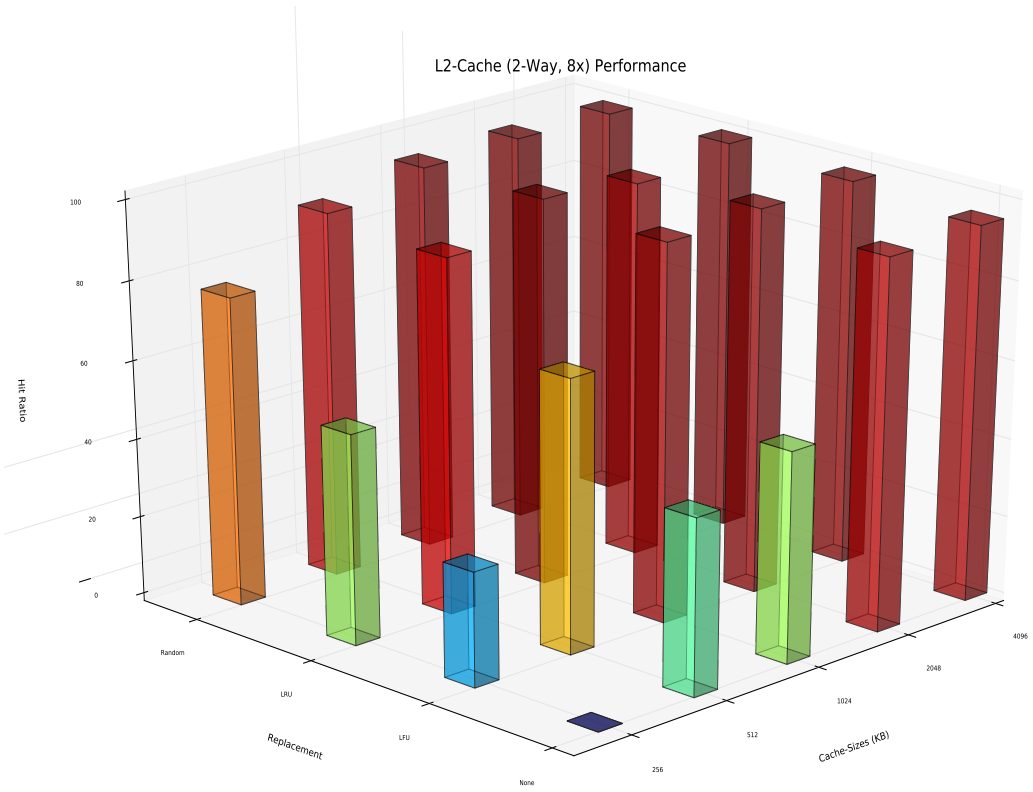


Figure E.28: Chess game L2-Cache designs Replacement-Size 3D plot

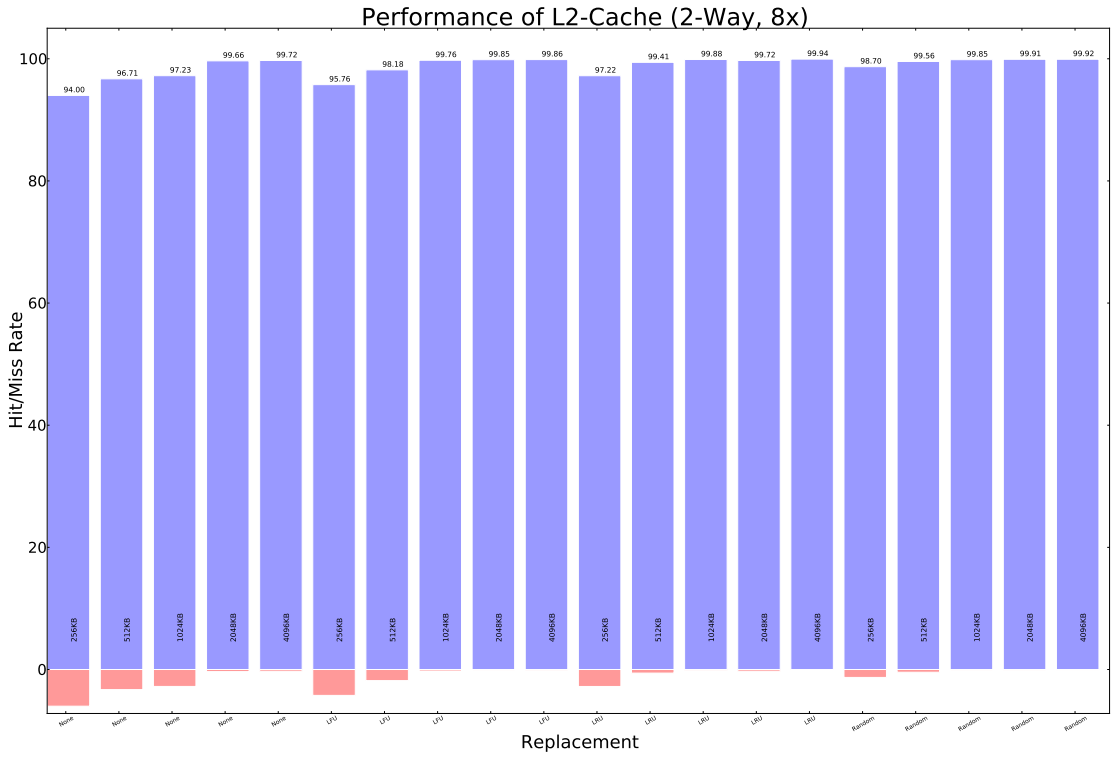


Figure E.29: Chess game L2-Cache designs Replacement-Size Bar plot

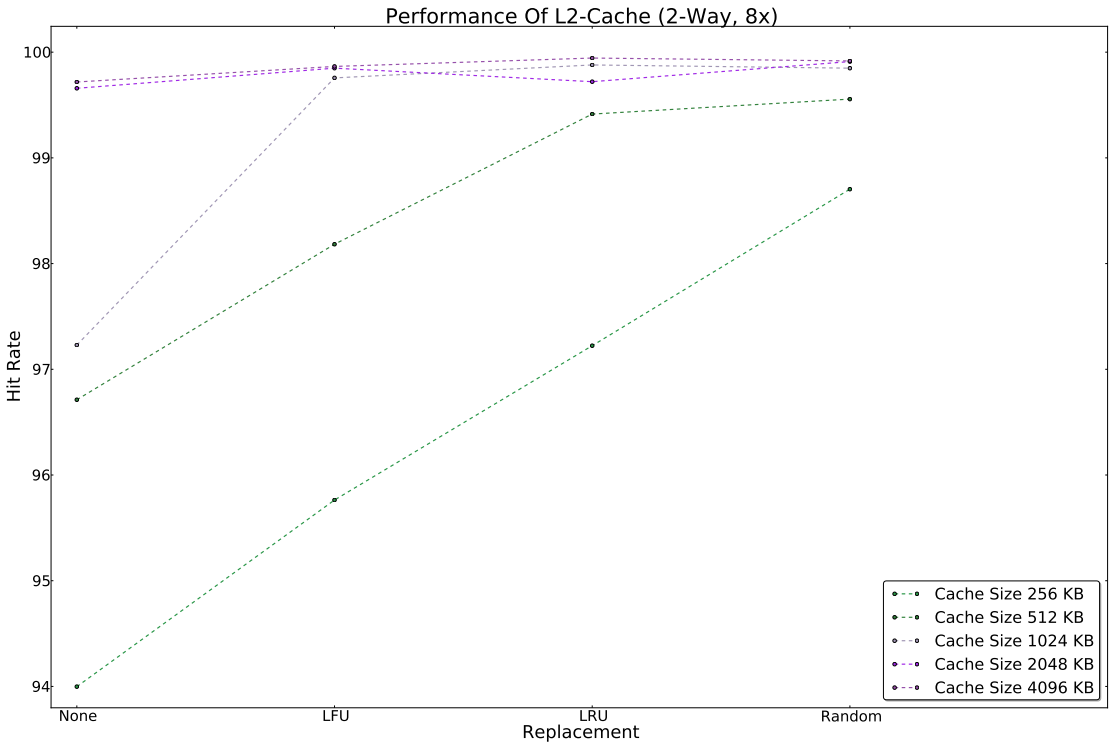


Figure E.30: Chess game L2-Cache designs Replacement-Size Line plot

References

[1] Virtual memory. URL http://ozark.hendrix.edu/~burch/csbsju/cs/350/notes/21/.

[2] Memory allocation. [Online] Available: https://developer.gnome.org/glib/2.28/glib-Memory-Allocation.html, 2005-2012.

[3] Qemu emulator user documentation. [Online] Available: http://wiki.qemu.org/download/qemu-doc.html, 2010.

[4] Qemu hacking. [Online] Available: https://github.com/qemu/qemu/blob/master/HACKING, 2013.

[5] Texinfo - the gnu documentation system. [Online] Available: http://www.gnu.org/software/texinfo/, 2013.

[6] Erik Andersen. Buildroot: making embedded linux easy. [Online] Available: http://buildroot.uclibc.org/, 2006-2013.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[8] Fabrice Bellard and the QEMU team. Qemu manual, 2013. URL http://wiki.qemu.org/Manual.

[9] N. Devillard. Gnuplot interfaces in ansi c. [Online] Available: http://ndevilla.free.fr/gnuplot/, 2012.

[10] David Money Harris and Sarah L. Harris. *Digital design and computer architecture.*

[11] B Iain McNally. Advanced computer architecture. URL http://users.ecs.soton.ac.uk/bim/notes/aca/index.php.

[12] *MIPS32 Architecture For Programmers Volume II: The MIPS32 Instruction Set.* MIPS Technologies, 1225 Charleston Road Mountain View, CA 94043-1353, document number: md00086 edition.

[13] Patterson and Hennessy. *Computer Organization and Design.* Morgan Kaufmann, 2009.

[14] Gurpur Prabhu. Interaction policies with main memory. URL http://www.cs.iastate.edu/~prabhu/Tutorial/CACHE/interac.html.

[15] sporkBomber. Memory hierarchy, 2013. URL http://www.cheatography.com/sporkbomber/cheat-sheets/sty-2013-final/.

[16] Dominic Sweetman. *See MIPS Run*. Morgan Kaufmann, 2010.

[17] Aditya Tandon. Implement a pipelined arm processor with cache memory. Technical report, University of Southampton, 2013.

[18] Colin Kelley Thomas Williams. Gnuplot. [Online] Available: http://www.gnuplot.info/, 2013.